Pointers and References

PIC 10B, UCLA

Pointers and References

The C++ programming language gives its users incredible control in how memory is managed. Although when writing simple user-defined data structures and when using templated classes like **std::vector** we do not need to worry about individual chunks of memory, it is important to understand what is going on “under the hood.” When we write more complicated classes that need to manage memory directly or wish to understand such classes, we need a working knowledge of pointers and different types of references; different types of constructors; and the means of managing memory safely.

Pointers

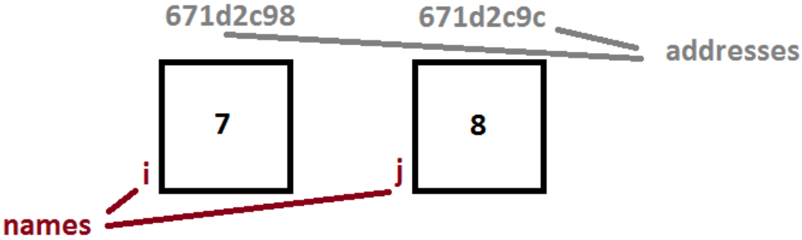
A pointer stores the **location** in memory of a particular **variable type.**
The value of a pointer is some hexadecimal (base 16) address.

When the pointer is dereferenced by the dereferencing operators
***** (or **->** to access member data),
we can directly access the variable stored at that address.

The address of a variable is accessed by the address-of operator **&.**

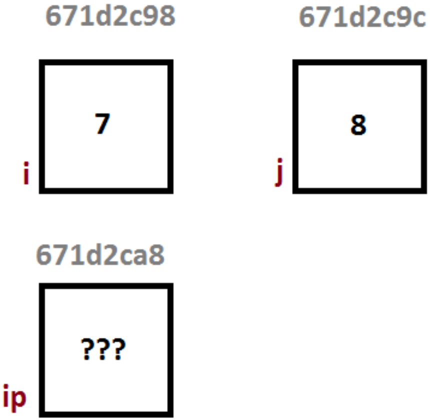
Pointers

int i=7, j=8;



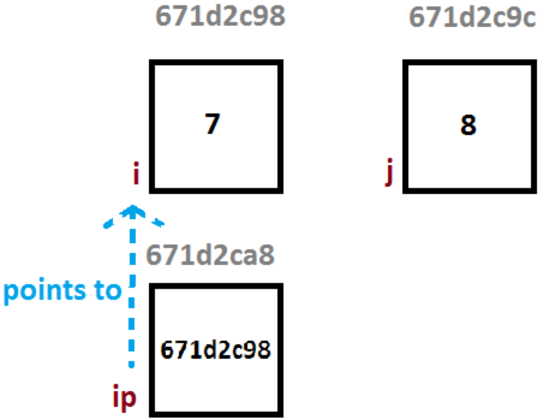
Pointers

int *ip; // ip is pointer to int, but points nowhere valid



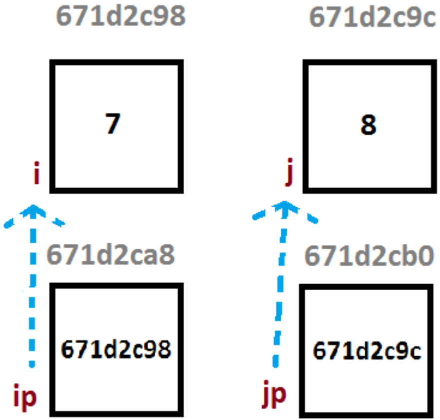
Pointers

ip = &i; // now ip == the address of i



Pointers

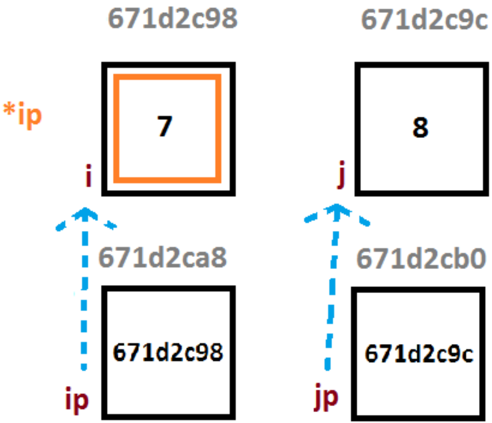
int *jp = &j; // jp is pointer to int, pointing to j



Pointers

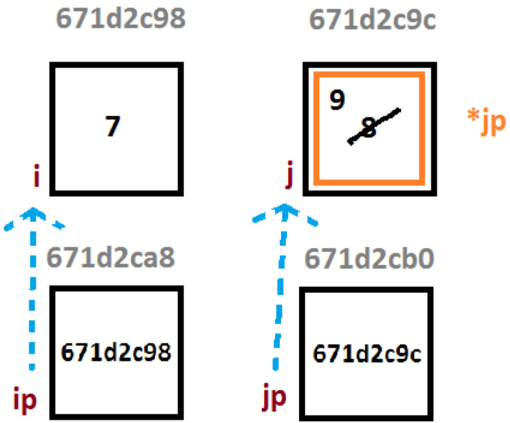
std::cout << *ip << " "; // prints 7 and space

7



Pointers

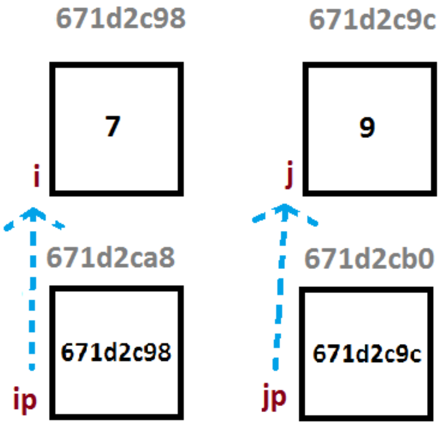
```
std::cout << (*jp)++ << " "; // prints 8 and now j==9
7 8
```



Pointers

std::cout << (*jp)++ << " "; // prints 8 and now j==9

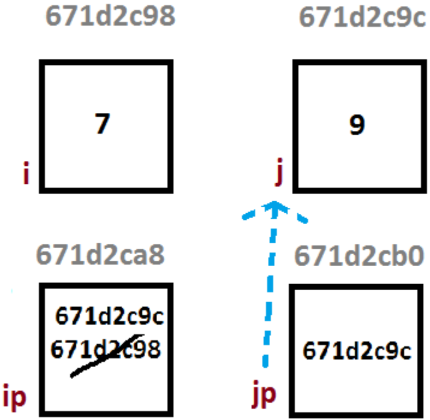
7 8



Pointers

ip = jp; // ip and jp both now point to j

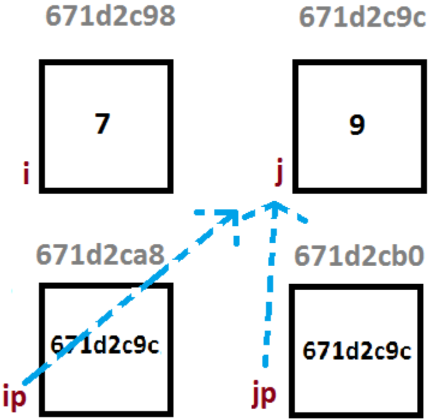
7 8



Pointers

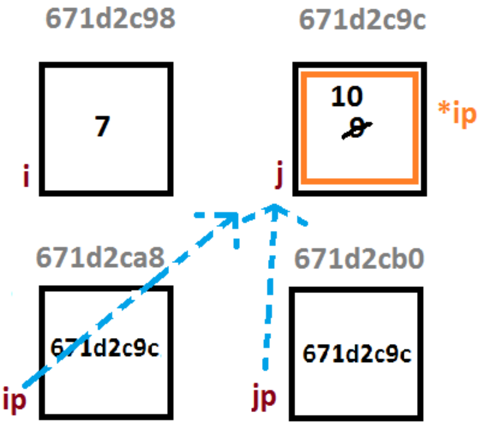
ip = jp; // ip and jp both now point to j

7 8



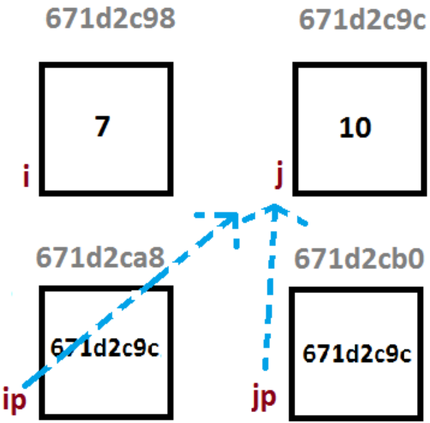
Pointers

```
++(*ip);
7  8
```



Pointers

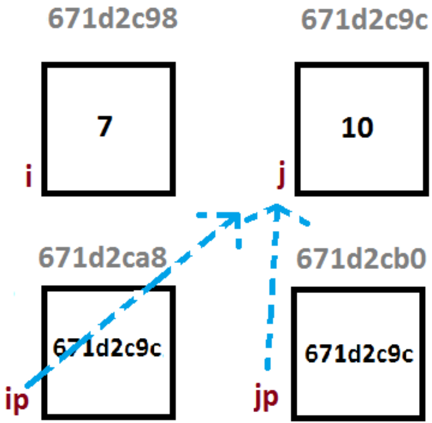
```
++(*ip);
7 8
```



Pointers

```
std::cout << j <<" " <<ip; // prints 10 and address of j
```

```
7 8 10 671d2c9c
```



Pointers

Pointers to objects can invoke member functions and member variables with **operator arrow**.

```
std::string s = "window";
std::string *sp = &s;

std::cout <<(*sp).size() <<'\n';
std::cout <<sp->size() <<'\n';
```

```
6
6
```

In general **(*x).** is the same as **x->** where **x** points to a class type.

Pointers

```
/* look for index where substring begins:
if found, get starting index;
if not found, get std::string::npos */

size_t thePos = sp->find("dow"); // thePos == 3

std::cout <<( (thePos != std::string::npos) ? "Found it!": "Not found!");

Found it!
```

Pointers

The type of object to which a pointer points cannot be changed and to assign a pointer a value, the value it is assigned to must match in the type it points to (except for **void*** to be discussed much later).

For example, the following is wrong:

```
double d = 3.14;
double *dp = &d;
std::string s("...");
std::string *sp = &s;

sp = dp; // ERROR: pointer to double vs pointer to string
sp = &d; // ERROR: assigning double pointer to pointer to string
*sp = 2.26; // ERROR: assigning a double to a string
int *ip2 = dp; /* ERROR: dp points to double, cannot be assigned to
   int pointer */
```

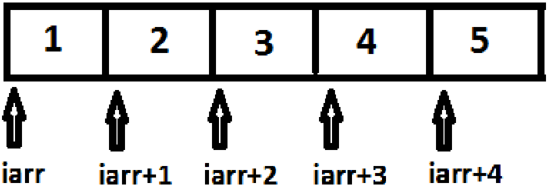
Pointers

When we allocate an array, the array variable name can be treated as a pointer!

We can dereference this pointer, perform arithmetic (increment/decrement, move forward n space or back m spaces), etc.

Generally **arr[i]** is the same as ***(arr+i)**.

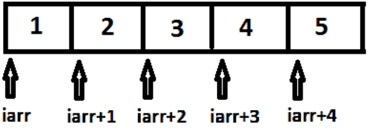
```
int iarr[5] = {1, 2, 3, 4, 5 };
```



Pointers

```
std::cout <<iarr <<" "; // outputs address of first (index 0) element

std::cout <<*iarr <<" "; // outputs first element

std::cout <<*(iarr+2) <<" "; // outputs third element (index 2) element

std::cout <<*iarr + 8 <<" "; // outputs 8 plus first element

int *ip = iarr; // ++advances (- - moves back)
++ip; // now ip points to second element
std::cout <<*ip <<" ";
```

```
c14eb424 1 3 9 2
```



Placement of const keyword

As a rule, the **const** keyword modifies the item *to its left*, unless it cannot modify anything to its left in which case it modifies the item to its right...

Many programmers, write code such as:

```
const double d = 7.73;
const std::string& s = foo();
```

etc. But it would be equally valid to write:

```
double const d = 7.73;
std::string const & s = foo();
```

Const and Pointers

From these "grammatical" insights into **const**, we consider different types of pointers and their constness.

```
int x[] = { 42, 43, 44 };

int *xp = x; // xp points to 42
const int *yp = x; // yp points to 42
```

- **xp** is a regular pointer to int, **int***: we can change where **xp** points, or modify the value ***xp**, i.e., what it points to, without problem.

  ```
  ++xp; // valid
  *xp = 123; // fine
  ```

- **yp** is a pointer to a constant int, **const int***: we can change where **yp** points, but we cannot modify the value ***yp**.

  ```
  ++yp; // valid
  *yp = 0; // ERROR: yp points to const int!
  ```

Const and Pointers

```
int x[] = { 42, 43, 44 };
int *xp = x; // xp points to 42

int * const zp = xp; // zp points to 42
const int * const wp = x; // wp points to 42
```

- **zp** is a *constant pointer* to an int, **int * const**: we can change the value ***zp**, i.e., the value it points to, but we cannot change where **zp** points.

  ```
  *zp = 0; // okay
  ++zp; // ERROR: zp is a const pointer
  ```

- **wp** is a constant pointer to a constant int, **const int * const**: we cannot change where **wp** points nor can we change the value ***wp** through **wp**.

  ```
  *wp = 3; // ERROR: wp points to const int
  ++wp; // ERROR: wp is a const pointer
  ```

Const and Pointers

Consider:

```
int i = 322;
const int *ip = &i;
```

We cannot do something like:

```
// ERROR: ip points to const int, cannot use it to point to int
int *ip2 = ip;
```

The above would *indirectly give the ability to modify* **i** *through* **ip**, *but* **ip** *treats* **i** *as* **const**...

References

An **object** is a segment of memory storing some bits: often the term "object" in C++ refers to any entity, be it a class object or a fundamental type like **int**, **char**, etc.

A **reference** is an entity that is bound to a particular object, as long as that reference exists.

In practice, we often think of a reference as giving another name for a variable, such that the reference name and the variable name are interchangeable.

References

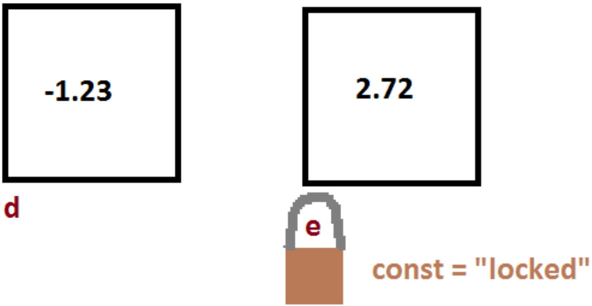
References must be *defined and initialized simultaneously*. As they are initialized, we often say that "bind" to a particular object.

With the exception of *reference to const*, an lvalue reference (reference with a single &) can only bind to an object with a location in memory.

Remark: it can be helpful to think of a reference as an *automatically dereferenced constant pointer*. The reference's location cannot change (like how the location a constant pointer points to cannot change) but we do not need any *****'s or **->**'s to access what the reference is referencing.

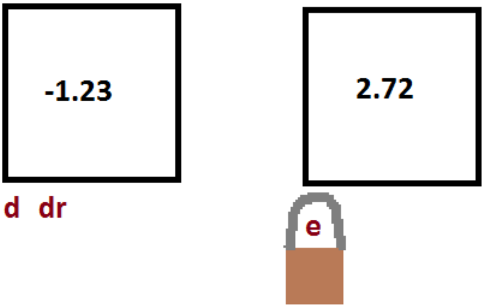
Lvalue References

double d = -1.23; // regular double
const double e = 2.72; // a const double



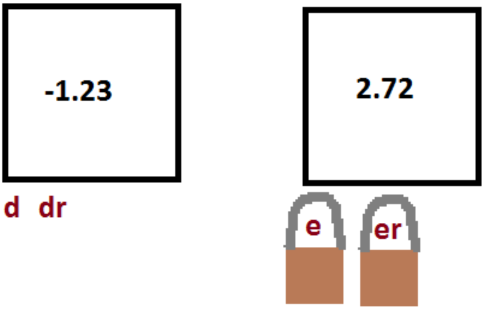
Lvalue References

double &dr = d; // dr references d



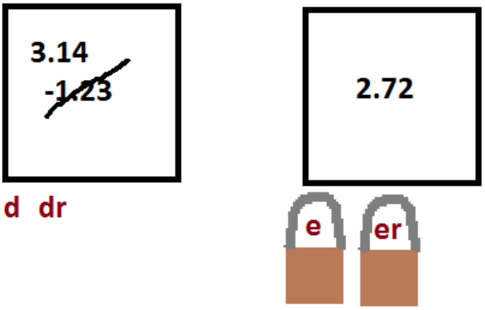
Lvalue References

const double &er = e; // er references e as const



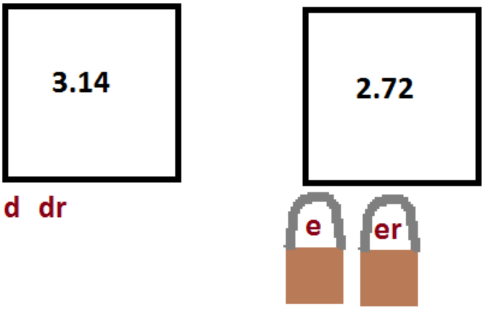
Lvalue References

dr=3.14; // changes both d and dr!



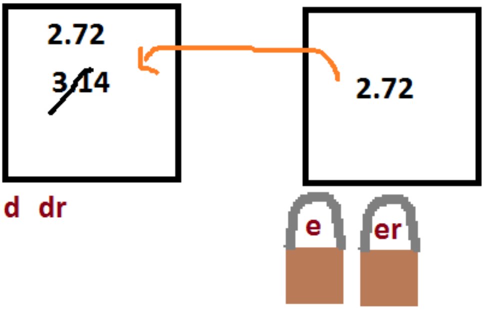
Lvalue References

```
dr=3.14; // changes both d and dr!
```



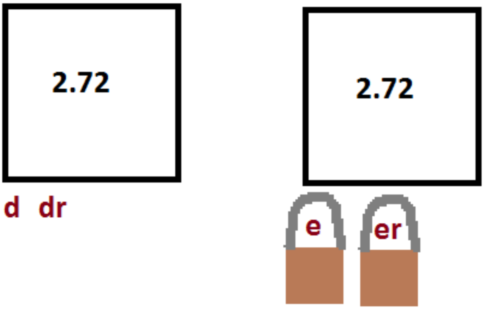
Lvalue References

dr = er; // changes both d and dr, no change to e or er



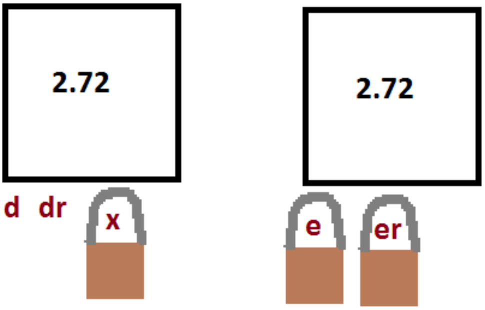
Lvalue References

dr = er; // changes both d and dr, no change to e or er



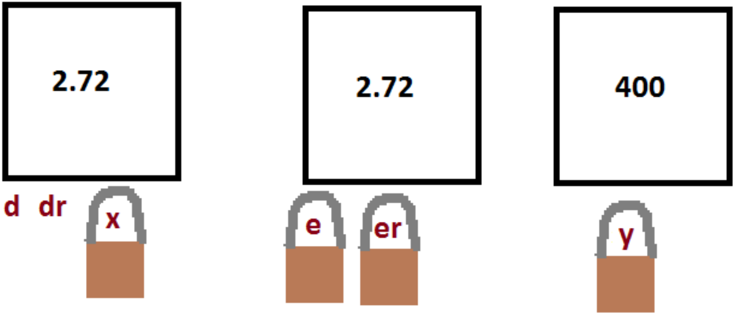
Lvalue References

const double& x = d; // x references d as const



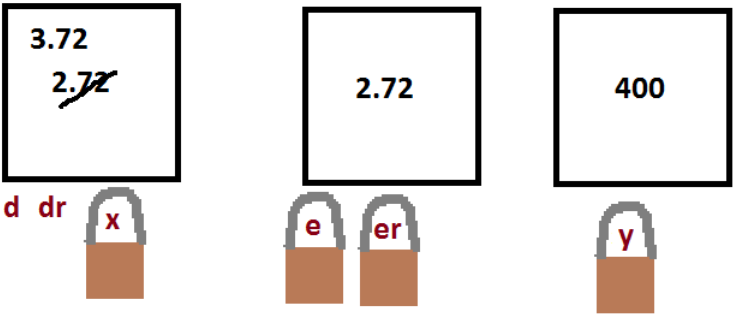
Lvalue References

// y references new block of memory as const, value of 400
const double&y = 400;



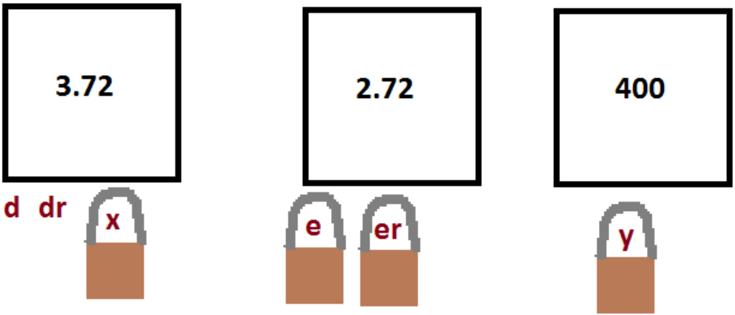
Lvalue References

++dr; // change d, dr, and x through dr!



Lvalue References

++dr; // change d, dr, and x through dr!



Lvalue References

The following lines of code are erroneous:

```
/* cannot make reference to double from const double or reference to
const double */
double & a = e; // e is const double
double& a2 = er; // er is reference to const double

int j = 22;
double &k = j; // cannot bind double reference to an int: type mismatch

++x; /* cannot do this as x is a reference to const (even though it
   refers to the non-const d) */
```

Lvalue References

An **lvalue reference to const** can bind to an rvalue (a temporary value that may not have a location in memory), but a normal **lvalue reference** cannot:

```
int& z = 13; // ERROR

const int& w = 13; // okay
```

Lvalue References

Basically:

- We cannot bind an lvalue reference to an rvalue.
- We cannot bind a normal (non-const) reference to something that is **const** because that *indirectly gives permission to modify the const object through that reference*.
- We cannot modify an object, even if it is not really const, through a reference to const.

Const References

Remark: effectively all references are const! A reference is bound to a given segment of memory just like a *constant pointer*.

The proper term for a variable like **y** below is **reference to const**, not a "const reference".

```
int x = 4;
const int& y = x;
```

Syntax & and *

Remark: when a variable is **first defined**, pairing it on the left with

- **&** makes it an lvalue reference, and
- * makes it a pointer.

Once a variable **has already been defined**, using

- **&** on its left returns its address, and
- * on its left dereferences the variable.

```
double d = 0;
double &x = d; // x is a REFERENCE to d
double *y = &x; // y is a POINTER to x (and d), &x is the address of d
++(*y); // *y IS d
std::cout <<d; // prints 1
```

Syntax & and * I

Consider **Cat** and a **Person** classes showing how people are owned by cats and cats have people as servants... close enough to reality.

The classes store pointers to one another. Compilers only allow a symbol to be used after it has been declared: observe the first line of declaring a **Person** class (more on declarations/definitions later). Just like a function is *declared* by giving a signature, no body required, a class is *declared* by listing it as a class, no interface required.

Syntax & and * II

```
class Person; // declare that Person is a class

class Cat {
private:

   // since Person was declared, this is okay
   const Person *servant = nullptr;

public:

   // _servant is a pointer to const, can be assigned to member
   void assign_servant(const Person* _servant) { servant = _servant; }

   // p is a reference to const, &p is a pointer, can compare to member
   bool is_servant(const Person& p) const { return &p == servant; }
};
```

Syntax & and * III

```
class Person {
private:
   const Cat *owner = nullptr;
public:
   void assign_owner(const Cat* _owner) { owner = _owner; }
   bool is_owner(const Cat& c) const { return &c == owner; }
};
```

So each class stores a **pointer to const** for the other class, initially being **nullptr**.

Note how the **assign_** functions accept a pointer (as pointer to const) and do pointer assignment, while the **is_** functions accept a reference (as reference to const) and compare addresses!

Const correctness applies with pointers, too!

Syntax & and * IV

Here we use those classes and functions:

```
Cat cotton;
Person patricia;

cotton.assign_servant(&patricia); // accepts a pointer!
patricia.assign_owner(&cotton); // accepts a pointer!

cotton.is_servant(patricia); // true
```

Initialization and Assignment

Multiple variables can be defined (and initialized) as a single statement. Initialization is a left-to-right operation.

```
std::string *sp1 = nullptr, s1("hello"), &sr1 = s1,
   *sp2 = &s1, s2, &sr2 = *sp2;

// sp1 is a pointer to string, pointing to null
// s1 is a string, initialized to "hello"
// sr1 is a reference to a string, bound to s1
// sp2 is a pointer to string, pointing to s1
// s2 is a string, default initialized to ""
// sr2 is a reference to a string, bound to where sp2 points
```

Basically we include the pure type on the left and we need to add a * or **&** to the left of a variable to make it into a pointer or reference to the type on the left.

Initialization and Assignment

Multiple variables can be assigned to the same value with the **assignment operator =.** Assignment is a right-to-left operation.

```
s1 = s2 = "new message";
/* s2 is assigned to "new message" and its new value is returned and
   used to assign s1 to the same value */
```

Const Casts

There is a reason the **const** keyword exists. A variable is not supposed to change if it is **const**!

Using a **const_cast** to modify constness almost always indicates a design flaw and lack of good coding practices or judgment. It does have other uses, too, relating to the **volatile** keyword.

From a pedagogical standpoint, it's important to know of the existence of a **const_cast.** This sort of cast can work on pointers and references.

Const Casts

```
double d = 0; // so d is not const
const double& dr = d; // but dr is a reference to const
const double *dp = &d; // and dp is a pointer to const...

++dr; // nope!
double *dp2 = dp; // not allowed!

++const_cast<double&>(dr); // reference to dr without the const, d==1

/* cast away constness so have regular double*, can initialize dp2 from
   that */
double *dp2 = const_cast<double*>(dp);
```

Don’t forget if the underlying variable really is const, we face **undefined behaviour** in changing it.

d isn’t really const, though...

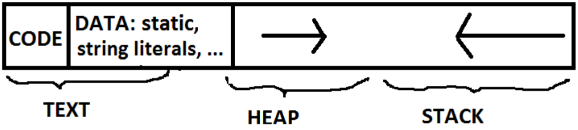
Pointer vs Reference

Pointer	Reference
- stores a variable address and type	- gives another name for a variable or block of memory
- reassignment changes pointed-to location	- reassignment changes the value stored in the referenced memory segment but not the memory location the reference references
- must be dereferenced to modify pointed-to variable	- already dereferenced
- can be defined without initialization	- must be defined and initialized together
- can point to null	- cannot reference null

Text, Stack, and Heap

When the a program is run, the loader allocates memory in the RAM for the operations of the program. This includes regions known as the **text segment**, the **heap**, and the **stack**.

Within the text segment are found the code memory (the raw machine code for the program to operate) of lowest address and the data of higher address. The data stores the static and global variables: the variables that have memory allocated for them or are initialized before **int main()** runs.



Stack

The stack stores local variables created during the running of the program and, in the case of functions, the return address where to send the return value, etc.

The stack observes a last-in-first-out behaviour (LIFO) similar to that of a physical stack: as items are added to a stack, the item at the top must also be the first removed. When a variable goes out of scope, it is removed from the stack.

Stack variables can be quickly accessed by the CPU and do not require the use of pointers. We use stack variables when we know how many variables, i.e., how much memory, we need to begin with.

Stack

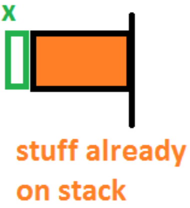
Recall that a set of braces defines a scope:

```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```

Stack

Recall that a set of braces defines a scope:

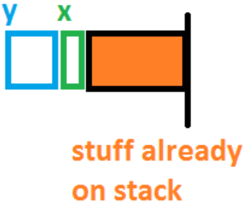
```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

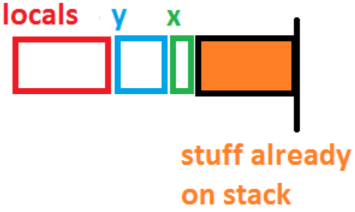
```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

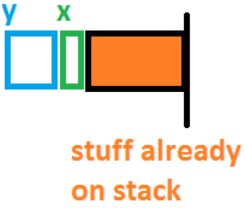
```
{
    int x = 4;
    {
        double y;
        y = std::sin(x); /* The sine function (probably) creates
            local variables to capture x to do computation */
    } // y goes out of scope and is destroyed
    double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

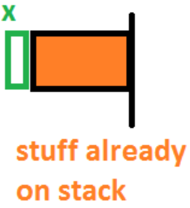
```
{
    int x = 4;
    {
        double y;
        y = std::sin(x); /* The sine function (probably) creates
            local variables to capture x to do computation */
    } // y goes out of scope and is destroyed
    double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

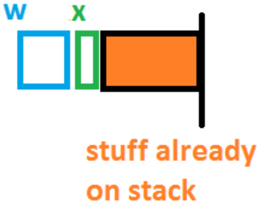
```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

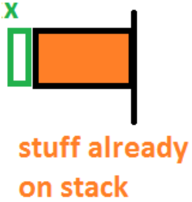
```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```



Stack

Recall that a set of braces defines a scope:

```
{
   int x = 4;
   {
      double y;
      y = std::sin(x); /* The sine function (probably) creates
         local variables to capture x to do computation */
   } // y goes out of scope and is destroyed
   double w = 0;
} // x and w go out of scope: first w is destroyed, then x
```

stuff already
on stack

Heap

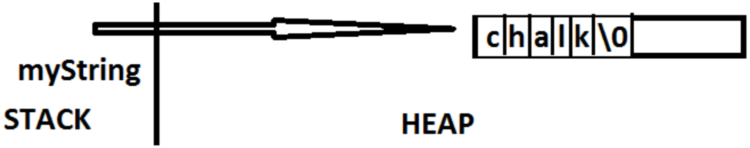
The heap (also called **“free store”**) stores objects in so called **dynamic memory.** This is useful when we don’t know exactly how much memory we may require (so memory allocation may need to be dynamic).

Data access is slower for heap memory, and we can only access heap memory through pointers (i.e. knowing the memory addresses).

Heap

When we use container classes such as **std::string**, **std::vector<T>**, **std::set<S>**, etc., the local variables that we refer to live on the stack, but internally they store a pointer to heap memory.

std::string myString("chalk");



Remark: a string literal is a contiguous sequence of chars in the **data (text)** memory, terminated by a null character **'\0'**.

std::string also likely stores its data terminating by the null character.

new and delete

The keywords **new** and **delete** are used for dynamic memory storage.

The **new expression** allocates memory, creates an object (or fundamental type on the heap), and returns a pointer to that object.

The **delete expression** destroys the object (or fundamental type) to which a dynamic pointer points and it frees up that memory.

These are very delicate operations and using these operations without great care can cause serious bugs due to accessing invalid locations in memory and memory leaks (when heap memory is tied up but can no longer be accessed).

Modern C++ offers smart pointers to help with these dangerous memory management issues. We will look at both.

new and delete

When allocating heap memory, we can pass construction parameters.

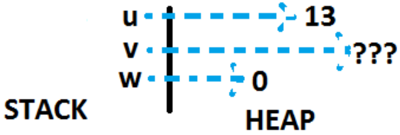
Without any parameters, default initialization is done: invoking default constructors and leaving fundamental types uninitialized!

With the () syntax, variables are value-initialized so that class types are default constructed and fundamental types are set to 0.

new and delete

```
// u points to unsigned int on the heap, value 13
unsigned *u = new unsigned(13);

// v points to unsigned int on heap, value unknown
unsigned *v = new unsigned;

// w points to unsigned int on heap, value 0
unsigned *w = new unsigned();
```

Note: u, **v**, and **w** are stack-based pointers! They are not themselves **unsigned int**s!
u, **v**, and **w** point to **unsigned int**s that are stored in the **heap.**



new and delete

```
// s points to std::string on heap, value "BBBBBBBBBB"
std::string *s = new std::string(10, 'B');

// s2 points to std::string on heap, value ""
std::string *s2 = new std::string;

// s3 points to std::string on heap, value ""
std::string *s3 = new std::string();
```

new and delete

The error of allocating heap memory but not releasing that memory when it is no longer needed is called a **memory leak.** This can lead to programs slowing down or crashing due to memory exhaustion.

When a pointer to dynamic memory goes out-of-scope, the pointer goes away (removed from the stack), but the heap memory remains.

new and delete

To properly free the memory, we need to request that from the compiler with the **delete** expression.

If **p** is a pointer then in writing

```
delete p;
```

we have told the compiler to free up the heap memory pointed to by **p.**

The C++ Standard makes no guarantees as to where **p** points after the call to **delete**.

new and delete

When using the **delete expression**, one must be careful.

First, it is an error to invoke delete on a non-pointer object. And the following generated *undefined behaviour:*

- deleting a pointer to a stack variable,
- deleting a pointer that has already been deleted and which is not set to **nullptr**, or
- to dereference a deleted pointer or **nullptr**.

new and delete

Recall **u, v, and w** pointed to **unsigned** values and **s, s2, s3** pointed to **std::string**s.

```
/* pointed-to unsigned values destroyed and memory blocks pointed to by
   u, v, and w are freed */
delete u;
delete v;
delete w;

/* strings pointed by by s, s2, s3 are destroyed and memory blocks
   pointed by by s, s2, and s3 are freed */
delete s;
delete s2;
delete s3;
```

new and delete

It is good practice to set a pointer to **nullptr** after it has been deleted. A statement such as **if (u)** will convert the pointer u to a bool: **true** if **u != nullptr** and **false** if **u == nullptr**.

```
u = nullptr; // u was deleted and now points to null

if (u) { // will only dereference u if still valid
   std::cout << *u << '\n';
}
```

Also, there is no harm in deleting it again once it is null:

```
delete u; // okay, does nothing
```

new and delete

Recall **v** and **s** are pointers that have been deleted but do not point to null. **u** has been deleted and now points to null.

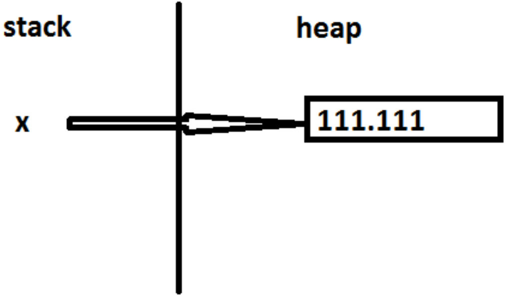
```
// cannot delete v again - v was already deleted but not set to nullptr!
delete v;

// cannot dereference deleted memory: s is called a dangling pointer
std::cout <<s->size();

std::cout <<*u; // cannot defererence nullptr

unsigned a = 7; // a is a stack variable
unsigned *b = &a; // b points to it
delete b; // deleting a stack variable is undefined
```

new and delete

```
{ // x goes out of scope before memory freed. Memory leak!
   double *x = new double(111.111);
}
```

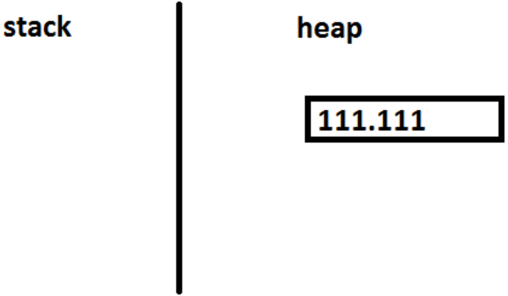
new and delete

```
{ // x goes out of scope before memory freed. Memory leak!
   double *x = new double(111.111);
}
```



new and delete

```
{ // x goes out of scope before memory freed. Memory leak!
   double *x = new double(111.111);
}
```

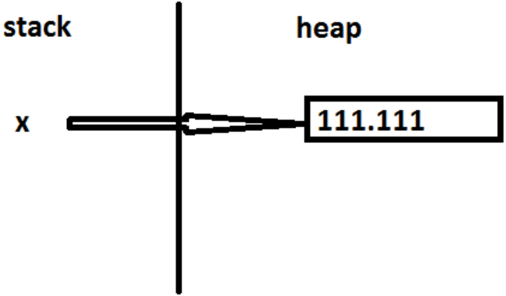


new and delete

```
{ // no memory leak: we freed the memory in time!
   double *x = new double(111.111);
   delete x;
   x = nullptr; // not necessary here but a good idea
}
```

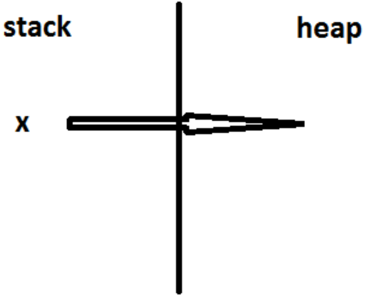
new and delete

```
{ // no memory leak: we freed the memory in time!
   double *x = new double(111.111);
   delete x;
   x = nullptr; // not necessary here but a good idea
}
```



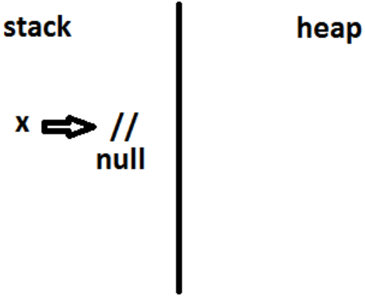
new and delete

```
{ // no memory leak: we freed the memory in time!
   double *x = new double(111.111);
   delete x;
   x = nullptr; // not necessary here but a good idea
}
```



new and delete

```
{ // no memory leak: we freed the memory in time!
   double *x = new double(111.111);
   delete x;
   x = nullptr; // not necessary here but a good idea
}
```



new and delete

```
{ // no memory leak: we freed the memory in time!
   double *x = new double(111.111);
   delete x;
   x = nullptr; // not necessary here but a good idea
}
```

new and delete

To allocate and clear contiguous blocks of memory, we use the **new []** and **delete []** expressions.

Note that the return of the **new []** expression is a pointer to the first element, not an array!

The array size parameter need not be a **const** value like for static arrays.

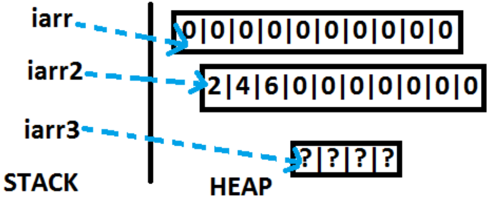
new and delete

With C-style arrays:

```
size_t sz = 42;
int iarr[sz]; // ERROR: sz is not constant!

// allocate a static array (on the stack)
constexpr size_t statSize = 100;
int iarr[statSize]; // okay with constant size
```

new and delete

```
size_t TEN = 10; // not declared const

int *iarr = new int[TEN](); // ten ints, all 0's
int *iarr2 = new int[TEN] {2, 4, 6 }; // ten ints, first 3 are 2, 4, 6; rest are 0's

// allocate some number of integers based on output of foo() function
int *iarr3 = new int[foo()]; // foo() ints, all uninitialized!

// three strings: "A", "B", and ""
std::string *sarr = new std::string[3] {"A", "B"};
```



new and delete

The **new []** expression is limited in what it can do for initial values.

The values in the contiguous block can be: default constructed (for fundamental types this means uninitialized), value initialized (so fundamental types would be set to 0); or, if the first several values are set by the user, the remaining values will be value initialized.

If a class lacks a default constructor, it cannot be **new[]**ed!

new and delete

```
delete [ ] iarr; // dynamic array is deleted and memory freed
iarr = nullptr;
```

Note: if **delete** is used instead of **delete []** for dynamic arrays, the behaviour can be undefined. Likewise if **delete []** is called instead of **delete**, the behaviour can also be undefined.

new and delete

It is **not possible** to initialize a contiguous block like

```
// want all values to be 10 initially: CANNOT do so
int* allTen = new int[100](10);

// want all values to be "AAA" initially: CANNOT do so
std::string* allAAA = new std::string[100](3,'A');
```

new and delete

To ensure memory is properly managed when we work with the raw pointers returned from **new** and **new[] expressions**, etc., we need to be sure to not "misplace" any pointers.

Losing track of where we allocated memory is a sure way to have memory leaks.

RAII

RAII (resource acquisition is initialization) is a programming technique that binds a resource (such as heap memory or a link to a file) to the lifetime of an object. When the corresponding object is properly implemented, it will acquire the resource during its construction and at the end of its lifetime, it will automatically release the resource.

RAII

Behind the scenes, a **std::string** manages heap memory. When a variable/object goes out of scope, it is destroyed. When a **std::string** is destroyed, it frees the heap memory that it manages.

```
{ // some scope
   std::string msg("hi");
} // end of scope, msg will be destroyed and the heap memory freed
```

```
{ // some scope
   char *msg = new char[3] { 'h', 'i', '\0' };
} // end of scope, memory leak has taken place!
```

There are also **smart pointers** that can be used like pointers, but which fulfill RAII.

Smart Pointers

Smart pointers are templated classes found in the **<memory>** header that manage heap memory automatically and guard against memory leaks.

A **shared_ptr** is an object that holds an object in heap memory and tracks the number of **shared_ptr** objects referencing that object. As the name and the previous sentence suggest, **shared_ptr** objects can share an object on the heap.

If the reference count of the number of **shared_ptr** objects pointing to a heap object drops to 0, that heap object is automatically destroyed and the memory is freed.

A **unique_ptr** is an object that allows only a single reference to a heap object.

Smart Pointers

Remark 1: one should be very careful about mixing **shared_ptr**s, **unique_ptrs**, and raw pointers.

Remark 2: given the choice between a **unique_ptr** and **shared_ptr**, the **unique_ptr** is preferable for efficiency since no reference count is required.

Remark 3: there are also **weak_ptr**s, but we do not discuss them.

Shared Pointers

The **make_shared** function constructs an object with its input construction parameters and returns a **shared_ptr**.

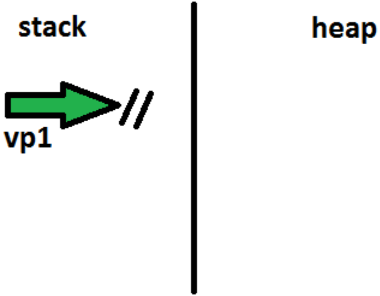
With default construction, a **shared_ptr** will point to null.

Shared Pointers

```
void does_nothing() {
    using intVec = std::vector<int>;
    std::shared_ptr<intVec> vp1; // vp1 == nullptr (default initialized)
    {
        // vp2 points to 3 zeros
        std::shared_ptr<intVec> vp2 = std::make_shared<intVec>(3);
        vp1 = vp2; // there are two objects pointing to the vector
    } // vp2 is out of scope so just one object points to the vector
} // the function is over, vp1 is out of scope, the memory is freed
```

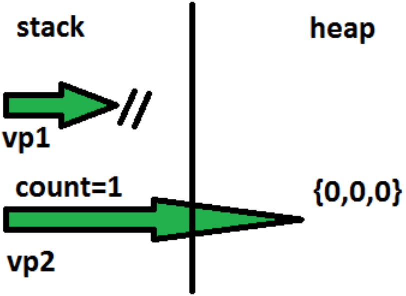
Shared Pointers

```
void does_nothing() {
    using intVec = std::vector<int>;
    std::shared_ptr<intVec> vp1; // vp1 == nullptr (default initialized)
    {
        // vp2 points to 3 zeros
        std::shared_ptr<intVec> vp2 = std::make_shared<intVec>(3);
        vp1 = vp2; // there are two objects pointing to the vector
    } // vp2 is out of scope so just one object points to the vector
} // the function is over, vp1 is out of scope, the memory is freed
```



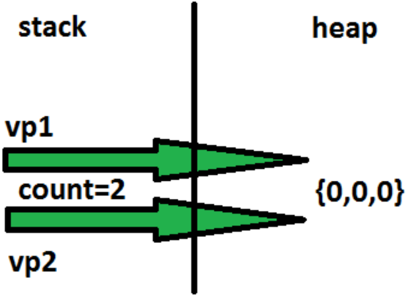
Shared Pointers

```
void does_nothing() {
   using intVec = std::vector<int>;
   std::shared_ptr<intVec> vp1; // vp1 == nullptr (default initialized)
   {
      // vp2 points to 3 zeros
      std::shared_ptr<intVec> vp2 = std::make_shared<intVec>(3);
      vp1 = vp2; // there are two objects pointing to the vector
   } // vp2 is out of scope so just one object points to the vector
} // the function is over, vp1 is out of scope, the memory is freed
```



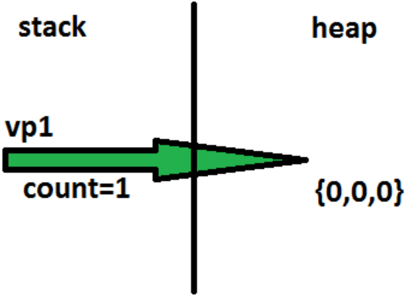
Shared Pointers

```
void does_nothing() {
   using intVec = std::vector<int>;
   std::shared_ptr<intVec> vp1; // vp1 == nullptr (default initialized)
   {
      // vp2 points to 3 zeros
      std::shared_ptr<intVec> vp2 = std::make_shared<intVec>(3);
      vp1 = vp2; // there are two objects pointing to the vector
   } // vp2 is out of scope so just one object points to the vector
} // the function is over, vp1 is out of scope, the memory is freed
```



Shared Pointers

```
void does_nothing() {
   using intVec = std::vector<int>;
   std::shared_ptr<intVec> vp1; // vp1 == nullptr (default initialized)
   {
      // vp2 points to 3 zeros
      std::shared_ptr<intVec> vp2 = std::make_shared<intVec>(3);
      vp1 = vp2; // there are two objects pointing to the vector
   } // vp2 is out of scope so just one object points to the vector
} // the function is over, vp1 is out of scope, the memory is freed
```



Shared Pointers

```
void does_nothing() {
    using intVec = std::vector<int>;
    std::shared_ptr<intVec> vp1; // vp1 == nullptr (default initialized)
    {
        // vp2 points to 3 zeros
        std::shared_ptr<intVec> vp2 = std::make_shared<intVec>(3);
        vp1 = vp2; // there are two objects pointing to the vector
    } // vp2 is out of scope so just one object points to the vector
} // the function is over, vp1 is out of scope, the memory is freed
```

stack	heap
count goes to 0	heap memory freed

Shared Pointers

Shared pointers can also be constructed by passing them outputs from **new** expressions, but the construction must be called **explicitly**, i.e., not given as the right-hand side during construction:

```
std::shared_ptr<int> sip( new int() ); // compiles
```

```
std::shared_ptr<int> sip2 = new int(); // ERROR!
```

Shared Pointers

Shared pointers also support other functions such as:
- **std::swap**, to switch the object pointed to by the shared pointers;
- the dereferencing operator *****;
- operator arrow **->**; and
- the implicit conversion of a shared pointer to boolean (true if not the null pointer).

Shared Pointers

The **reset** function relinquishes ownership of the pointed-to object and either points to null (with no arguments given) or to a new object (with a raw pointer argument given). For example:

```
std::shared_ptr<int> s( new int() );
s.reset( new int(2) ); // s now points to 2, no longer manages the 0
s.reset(); // s now points to nullptr, no longer manages to 2
```

Remark: a shared pointer can also point to a dynamically allocated array, but the construction is somewhat complicated and will be addressed when discussing templates.

Shared Pointers

```
auto dp( std::make_shared<double>(3.14) ),
   dp2( std::make_shared<double>() );
std::cout << *dp <<" " <<*dp2 <<'\n';

std::swap(dp,dp2); // the pointers have been swapped
if(dp2 && (*dp2 != 0)){ // if dp2 not nullptr and the number is nonzero
   std::cout <<"!!!"<<'\n';
}
std::cout <<*dp <<'\n';

dp.reset( new double(30.3) ); // dp no longer points to 0 but to 30.3
std::cout <<*dp;
```

```
3.14 0
!!!
0
30.3
```

Shared Pointers

Recall that the logical operators && and || are evaluated lazily: for &&, the first false encountered results in a value of **false**; for ||, the first true that is encountered results in an evaluation of **true.**

Unique Pointers

A **unique_ptr** cannot be copied or assigned-to another **unique_ptr** unless the variable used to do the assignment/initialization is at the end of its lifetime.

They are constructed in similar ways to **shared_ptr**s, by giving a pointer to heap memory or with **std::make_unique**.

The dereferencing operator applies, as does the conversion to a bool, **->**, etc..

Unique Pointers

The **std::swap** library function can also swap two unique pointers.

The **reset** function either removes the reference to an object or changes where the unique pointer points, depending on whether it is given a pointer argument or no arguments.

The **release** function *relinquishes control of the heap object* pointed to and returns its raw (not smart) pointer, with the unique pointer pointing to null. For example:

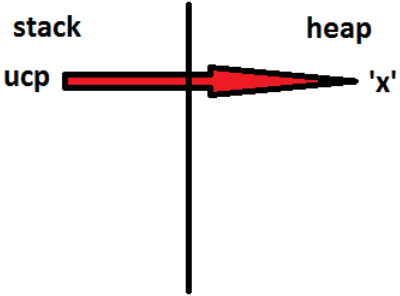
```
std::unique_ptr<int> u = std::make_unique<int>(3); // u manages the 3

// dangerous since i is just a raw pointer... u points to null
int *i = u.release();
delete i;
```

Unique Pointers

```
auto ucp(std::make_unique<char>('x')); // points to the char x
std::unique_ptr<char> ucp2; // null
ucp2.reset(new char('y')); // points to new char y
ucp2.reset(ucp.release()); // ucp points to null, ucp2 now points to x
ucp2.reset(); // points to null
```

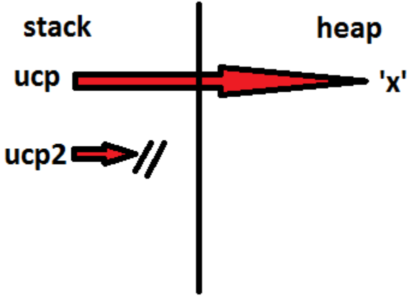
Unique Pointers

```
auto ucp(std::make_unique<char>('x')); // points to the char x
std::unique_ptr<char> ucp2; // null
ucp2.reset(new char('y')); // points to new char y
ucp2.reset(ucp.release()); // ucp points to null, ucp2 now points to x
ucp2.reset(); // points to null
```



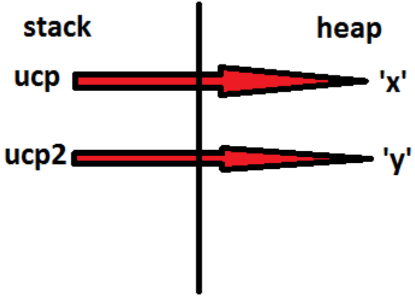
Unique Pointers

```
auto ucp(std::make_unique<char>('x')); // points to the char x
std::unique_ptr<char> ucp2; // null
ucp2.reset(new char('y')); // points to new char y
ucp2.reset(ucp.release()); // ucp points to null, ucp2 now points to x
ucp2.reset(); // points to null
```



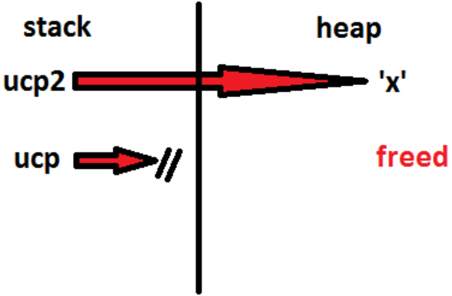
Unique Pointers

```
auto ucp(std::make_unique<char>('x')); // points to the char x
std::unique_ptr<char> ucp2; // null
ucp2.reset(new char('y')); // points to new char y
ucp2.reset(ucp.release()); // ucp points to null, ucp2 now points to x
ucp2.reset(); // points to null
```



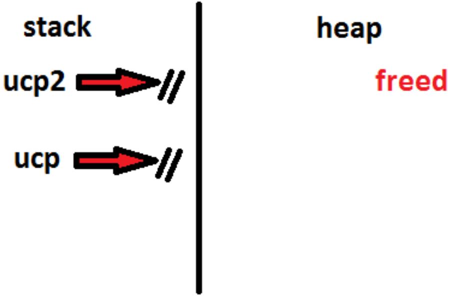
Unique Pointers

```
auto ucp(std::make_unique<char>('x')); // points to the char x
std::unique_ptr<char> ucp2; // null
ucp2.reset(new char('y')); // points to new char y
ucp2.reset(ucp.release()); // ucp points to null, ucp2 now points to x
ucp2.reset(); // points to null
```



Unique Pointers

```
auto ucp(std::make_unique<char>('x')); // points to the char x
std::unique_ptr<char> ucp2; // null
ucp2.reset(new char('y')); // points to new char y
ucp2.reset(ucp.release()); // ucp points to null, ucp2 now points to x
ucp2.reset(); // points to null
```



Unique Pointers

```
// okay ...
std::unique_ptr<double> dp = std::make_unique<double>( 10.6 );

// ERROR: such initialization would entail copying dp
std::unique_ptr<double> dp2 = dp;
```

Because **dp** will continue to exist past the erroneous line above, we cannot use it to initialize **dp2** as that would mean the pointers have to share.

On the other hand, the **std::make_unique** returns a **std::unique_ptr** but because that pointer is at the end of its lifetime (it has no name and will not exist after that single line), we can use that output to define **dp.**

Unique Pointers

A **unique_ptr** can easily be constructed to store a pointer to a dynamically allocated array by adding the **[]** after the data type**.** The array elements **can only be accessed by the subscript operator**.

```
std::unique_ptr<int []> darr( new int[6]() );
darr[0] = dar[3] = 4;
for (std::size_t i=0; i < 6; ++i){
   std::cout <<darr[i];
}
```

```
400400
```

The **std::unique_ptr<type []>** does not allow for dereferencing with * or **->**.

And **std::unique_ptr<type>** does not allow for subscripting with **operator[]**.

Unique Pointers

The **std::make_unique** function can also be used for dynamic arrays but besides a default constructor, it can only accept a size parameter for the array.

```
// ptr will be a unique_ptr pointing to an array of 14 doubles set to 0
auto ptr = std::make_unique<double[]>(14);
```

Constness of Smart Pointers

There are different types of **const**ness of smart pointers, too. Here's the parallels between them:

shared_ptr<Foo> behaves like **Foo***
shared_ptr<const Foo> behaves like **const Foo***
const shared_ptr<Foo> behaves like **Foo * const**
const shared_ptr<const Foo> behaves like **const Foo * const**

Value and Type

In C++, all **expressions** (things that can be evaluated) have a **type** and **value** category.

type: basically what something is

```
int x = 42; // x is an int

const int *xp = & x; // xp is a pointer to const int

int& y = x; // y is an lvalue reference to int

nullptr; // nullptr is of type std::nullptr_t (a special fundamental type)

std::string().size(); // this is of type size_t
```

Value and Type

The **value category** gives us information about whether a variable/expression is temporary (will no longer exist after a given line of execution - **rvalues**) or permanent (is assigned a place in memory - **lvalues**).

Prior to C++11, expressions were either **lvalues** or **rvalues.**

```
int x; // x is an lvalue
double d = 3.14 - 6; // d is an lvalue, 3.14-6 is an rvalue

// s is an lvalue, std::string("hello") is an rvalue
std::string s = std::string("hello");
s[3]; // s[3] is an lvalue
```

Since C++11, there are more value categories, so we venture into these other categories now...

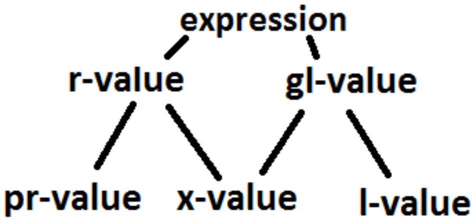
R-,GL-,PR-,X-, and L-Values

Expressions can be **rvalue**s or **glvalue**s, possibly both.

In turn, an **rvalue** can be either a **prvalue** (pure rvalue) or an **xvalue** (eXpiring value).

And a **glvalue** can either be an **lvalue** (sometimes “locatable” value) or an **xvalue**.

In the end, every expression is still either an **rvalue** or an **lvalue**.



R-,GL-,PR-,X-, and L-Values

A **prvalue** is generally something like a numeric literal or an unnamed temporary object like **std::vector<int>()**.

Generally we can say a prvalue does not have an addressable location in the RAM, perhaps simply being a value returned from a register during a computation.

C++ Standard: *"A prvalue is an expression whose evaluation initializes an object..."*

R-,GL-,PR-,X-, and L-Values

An **lvalue** is an object (or variable) with a permanent location in memory.

Often this means it can be assigned-to (unless it is const).

Very important rule: **if an entity has a name, it is an lvalue!**
Note: some entities don't have "names" but are still lvalues (like elements of an array/vector)!

Also, string literals are lvalues.

R-,GL-,PR-,X-, and L-Values

We often think of an **xvalue** as: it used to be an **lvalue** but became an **rvalue**, being cast from a more permanent state to a less permanent state.

Or: it used to be a **prvalue** but due to member access, it got promoted to an **xvalue**.

Unlike a **prvalue** that is not given a place in memory, an **xvalue** does exist in memory (even if that existence is only temporary).

R-,GL-,PR-,X-, and L-Values

Any output of the **std::move** function (to be defined) is an **xvalue**.

xvalue also includes objects that are prvalues upon which member access has been invoked.

And other things (it gets complicated)...

C++ Standard: *"An xvalue is a glvalue that denotes an object ... whose resources can be reused (usually because it is near the end of its lifetime)... Certain kinds of expressions involving rvalue references yield xvalues, such as a call to a function whose return type is an rvalue reference or a cast to an rvalue reference type."*

R-,GL-,PR-,X-, and L-Values

An **rvalue** is either a **prvalue** or an **xvalue**.

A **glvalue** is either an lvalue or an xvalue.

C++ Standard: *"A glvalue is an expression whose evaluation determines the identity of an object."*

R-,GL-,PR-,X-, and L-Values

The C++ standard offers little (clear) guidance on most of this. Here are a few excerpts for fun:

Whenever a prvalue appears in a context where a glvalue is expected, the prvalue is converted to an xvalue...

A prvalue of type T can be converted to an xvalue of type T. This conversion initializes a temporary object of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object.
[Example: struct X { int n; };
int k = X().n; // OK, X() prvalue is converted to xvalue —end example]

Note: An expression is an xvalue if it is ... a class member access expression designating a non-static data member of non-reference type in which the object expression is an xvalue, or ...

A function call is an lvalue if the result type is an lvalue reference type or an rvalue reference to function type, an xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise.

R-,GL-,PR-,X-, and L-Values

Consider the **bold underlined** expressions:

int x = **3 + 4**; // rvalue and prvalue: value from register

"abcdefg"; /* lvalue and glvalue: string literals (of type const char*) have memory locations */

std::vector<double> vdub { 1.1, 2.2 };
std::cout << **vdub[1];** /* glvalue and lvalue: each element has a place in memory */

vdub[0] *= 4; // glvalue and lvalue: subscript returns value by reference

((3<4) ? vdub[1] : vdub[0]) = 100; /* glvalue and lvalue: the ternary operator returns a value-type based on its return arguments. In this case returning vdub[1] (glvalue and lvalue) if 3<4 (true!) and otherwise vdub[0], which can then be assigned to (and hence must be an lvalue and glvalue */

R-,GL-,PR-,X-, and L-Values

double g = 9.8;
double *g = **&g**; // rvalue and prvalue

std::string word("rain");
std::move(word); /* xvalue, glvalue, and rvalue: the move function returns an x-value */

std::string("chocolate").substr(2); /* xvalue, glvalue, and rvalue: the unnamed string object begins as a pr-value but is converted to an xvalue because a member function is called upon it */

std::string("chocolate").substr(2); /* prvalue and rvalue: the substr function returns an unnamed string object */

R-,GL-,PR-,X-, and L-Values

```
// function signatures
std::string F();
const std::string& G(const std::string&);
std::string&& H(std::string&);

// using those functions...
std::string s = F(); // prvalue
G(s); // lvalue
H(s); // xvalue
```

R-,GL-,PR-,X-, and L-Values

Besides the “ordinary” lvalue references (&), there are also rvalue references (&&).

An **rvalue reference can only bind to rvalues.** They can help make various constructions and assignments more efficient**.**

An **lvalue reference can only bind to lvalues** but generally not r-values... **except for a reference to const.**

R-,GL-,PR-,X-, and L-Values

```
int &w = 5; // ERROR: cannot bind lvalue reference to pr-value!
```

```
int &&x = 5; // can bind rvalue reference to literal, pr-value
int &&y = x-7; // can bind to the prvalue result
std::string && s = std::string(); // can bind to the prvalue
```

One subtlety is that, above, **x**, **y**, and **s** are l-values after the initializations: *if is has a name, it has a spot in memory and is hence an l-value.*

```
int &&z = x; // ERROR: cannot bind rvalue reference to lvalue!
```

```
int &&w = std::move(x); // okay, bind rvalue reference to xvalue
```

R-,GL-,PR-,X-, and L-Values

std::move doesn't move anything! It merely acts as a cast.

However, the state of an object that has been moved from in constructing or updating another object is unknown. The object must be in a valid state, but the state is *unknown*.

Indeed, the containers of the C++ Standard Library have a special property that *unless otherwise specified, [...] moved-from objects shall be placed in a valid but unspecified state.*

R-,GL-,PR-,X-, and L-Values

The function **std::move** is is really just a cast to an rvalue reference:

```
std::string lvalue("L");

std::string s = std::move ( lvalue );

/* same thing is to write:
std::string s =
   static_cast< std::string && > ( lvalue ); */
```

But, the value stored in **lvalue** is unknown at this point.

```
lvalue.clear(); // okay, now ""
```

decltype

decltype has some similarities to **auto**, but
instead of deducing the type of the *right-hand-side* like **auto**,
it deduces the type of *its input argument.*

It is useful in working with lambda expressions and some template constructs (to come). For now, consider a function:

```
double foo(); // foo returns a double
```

Then:

```
decltype(foo()) x = 4; // x is double
const decltype(x) y = 8; // y is const double
```

decltype

```
// Recall: y is const double

decltype((y)) z = x; // z is const double& !!!!!!!!!!!!!!!!!!!
```

Skimming over many details here, the gist of **decltype** is:

- If an argument appears directly in the **decltype** parenthesis, **decltype** deduces the pure type of the argument; otherwise
- **decltype** deduces the value category instead: being **type** for a prvalue, **type&** for an lvalue, and **type&&** for an xvalue.

Since **x** is an lvalue, **decltype((x))** is a **type&** where **type&** happens to be **const double&** since **x** is a **const double**.

decltype

To quote the C++ Standard...

"For an expression e, the type denoted by decltype(e) is defined as follows:
— if e is an unparenthesized id-expression naming an lvalue or reference introduced from the identifier-list of a decomposition declaration, decltype(e) is the referenced type ...
— otherwise, if e is an unparenthesized id-expression or an unparenthesized class member access decltype(e) is the type of the entity named by e ...
— otherwise, if e is an xvalue, decltype(e) is T&&, where T is the type of e;
— otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;
— otherwise, decltype(e) is the type of e."

Namespaces

A namespace is effectively a scope where variables, classes, and functions can be defined. Many things are defined within the **std** namespace and we access elements of that namespace with the space name, the scoping operator **::**, and the element name. For example, **std::string**, **std::find_if**, **std::cout**, etc.

Defining things in a namespace can prevent cluttering up the “global namespace” with many variables/classes/functions with the same name. It also helps to disambiguate between classes/variables/functions defined in different libraries.

Namespaces

The basic syntax to write a namespace is

```
namespace nameOfSpace {
   /* stuff */
}
```

where **stuff** can be declarations and/or definitions.

We can also piece namespaces together: perhaps declaring elements in a header file and defining those elements in a cpp-file. To define, we:

- "open" the namespace when we want to use it, or
- use the scoping **::** operator.

Later to use the namespace elements, we

- use the scoping **::** operator.
- do a **using namespace our_name** to force compliance with our given namespace conventions (but only in a **.cpp**-file).

Namespaces

Header.h

```
#ifndef _EXAMPLE_
#define _EXAMPLE_

namespace example {

    // defining Y, declaring Y::foo and Y::bar
    struct Y {
        void foo() const;
        void bar(const Y&) const;
    };

    void baz(const Y&); // declaring baz
}

#endif
```

Namespaces

Y.cpp

```
#include<iostream>
#include "Header.h"

namespace example { // open up the namespace to define stuff
    void Y::foo() const { std::cout <<"foo"; }
}

// or just be specific with scope
void example::Y::bar(const Y& y) const {
    y.foo();
}
```

Within the **example::Y** class, **Y** is known so we don’t need to write

example::Y::bar(const <u>example::</u>Y& y)

Namespaces

baz.cpp

```
#include "Header.h"

void example::baz(const example::Y& y) {
   y.bar(y);
}
```

Namespaces

main.cpp

```
#include "Header.h"

int main() {
   example::Y y; // create a Y object as defined in example namespace
   example::baz(y); // calls Y::bar to call Y::foo to print "foo"
   return 0;
}
```

Constructors and Destructors

A **constructor** creates a class object and a **destructor** specifies what to do when the object has reached the end of its lifetime.

If no constructors are written, the compiler generates a **default constructor** for us (one taking no arguments) and attempts to default initialize everything; otherwise no default constructor is automatically generated.

Once we write any constructor for a class, we have taken responsibility for most of its construction processes.

If nothing is specified for a destructor, a default set of instructions are carried out to free the memory taken by the variables of a class (but this does not automatically free the heap memory when smart pointers are not used!).

Constructors and Destructors

There are two other constructors that are often compiler generated, but which we can write ourselves: the **copy constructor** and **move constructor**.

Copy constructors allow us to construct a new object by copying another object.

Move constructors allow us to construct a new object by harvesting the resources of an rvalue (either prvalue or xvalue, thus objects about to be destroyed).

Constructors and Destructors

Consider the code:

```
std::string b("Bob Foo");
std::string b_copy = b; // this is an independent copy of b
std::string b_copy2(b); // this is an independent copy of b
```

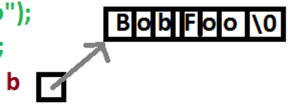
The variables **b_copy** and **b_copy2** are **copy constructed from b**. The variable **b** is unchanged through these constructions.

A copy constructor makes a newly constructed object into a copy of another.

Constructors and Destructors

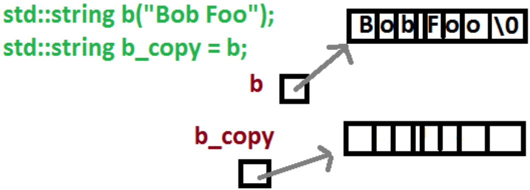
Depiction of what could happen with the copy constructor:

```
std::string b("Bob Foo");
std::string b_copy = b;
```



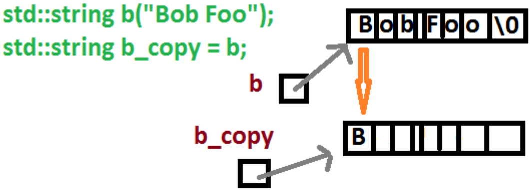
Constructors and Destructors

Depiction of what could happen with the copy constructor:



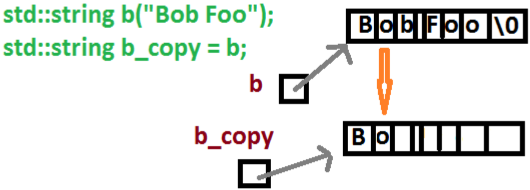
Constructors and Destructors

Depiction of what could happen with the copy constructor:



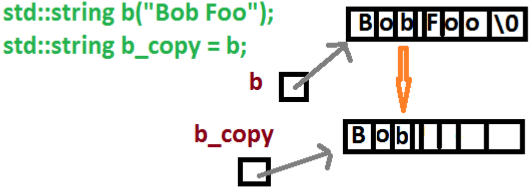
Constructors and Destructors

Depiction of what could happen with the copy constructor:



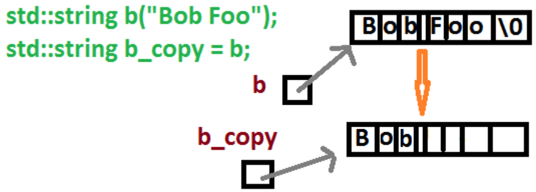
Constructors and Destructors

Depiction of what could happen with the copy constructor:



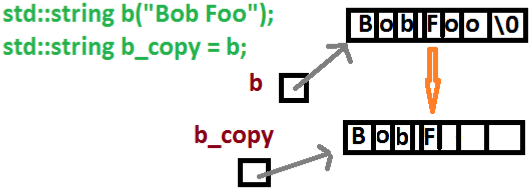
Constructors and Destructors

Depiction of what could happen with the copy constructor:



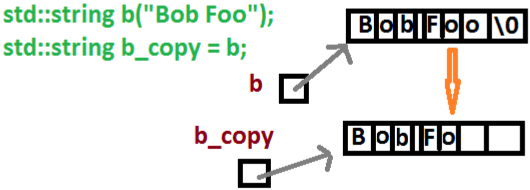
Constructors and Destructors

Depiction of what could happen with the copy constructor:



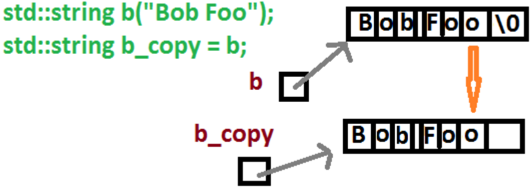
Constructors and Destructors

Depiction of what could happen with the copy constructor:



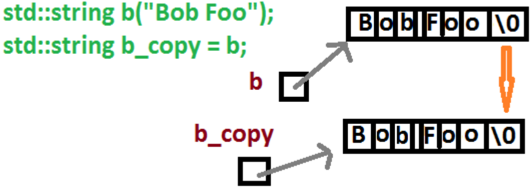
Constructors and Destructors

Depiction of what could happen with the copy constructor:



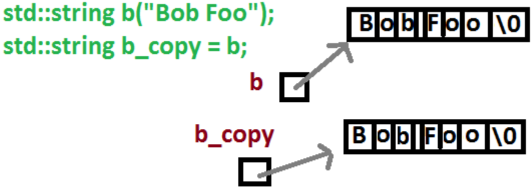
Constructors and Destructors

Depiction of what could happen with the copy constructor:



Constructors and Destructors

Depiction of what could happen with the copy constructor:



Constructors and Destructors

Suppose **make_name** is a function defined by:

```
std::string make_name() { return "Alice Bar"; }
```

Consider the code:

```
std::string name = make_name();
std::string name2(make_name());
```

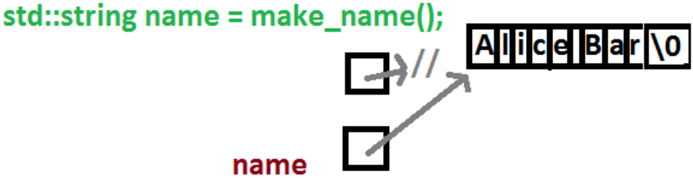
Since the outputs of the function **generate_name** are prvalues, it is possible to be more efficient in how **name** and **name2** are constructed. This can involve the **move constructor**.

Constructors and Destructors

Depiction of what could happen with the move constructor:

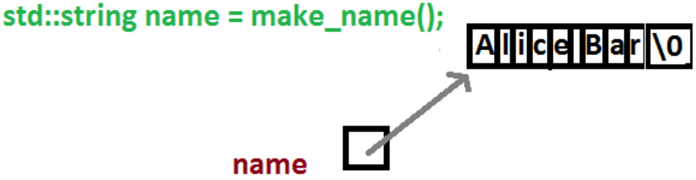
Constructors and Destructors

Depiction of what could happen with the move constructor:



Constructors and Destructors

Depiction of what could happen with the move constructor:



Constructors and Destructors

For the classes that we write, the compiler-generated copy and move constructors *memberwise* (i.e. variable-by-variable) *copy or "move"* data from the constructed-from object to create the new object.

Constructors and Destructors

The signatures of these constructors and the destructor are below for a class **T**:

```
T ( ); // default constructor

T ( const T & ); // copy constructor

T ( T && ); // move constructor

~T ( ); // destructor
```

Constructors and Destructors

The **default constructor**, if provided, should initialize all class variables to some sensible default values. It does not always make sense to have a default constructor!

The **copy constructor** should copy all the values stored in the assigned-from object: care is needed when dealing with heap memory because *we generally want a separate, independent copy.*

The **move constructor** generally takes the pointers and values from the constructed-from object, giving them to the constructed object, and leaves that constructed-from object in a state suitable for destruction.

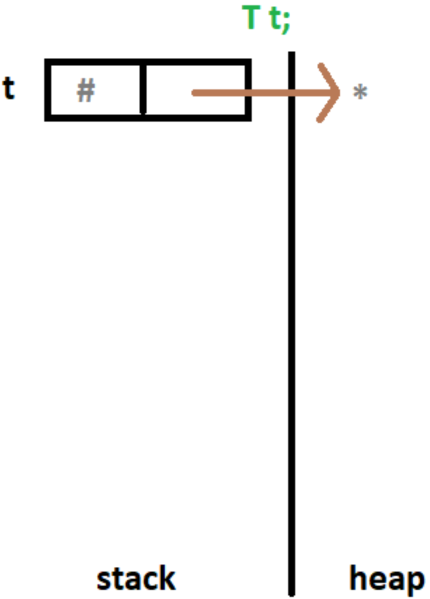
The **destructor** should ensure all the resources of the object are cleared up. When the new expressions have been used, there should be calls to delete expressions when raw pointers are being used.

Constructors and Destructors

In the following slides, we imagine that **T** is a class type that stores some memory directly on the stack and also stores a pointer that manages heap memory.

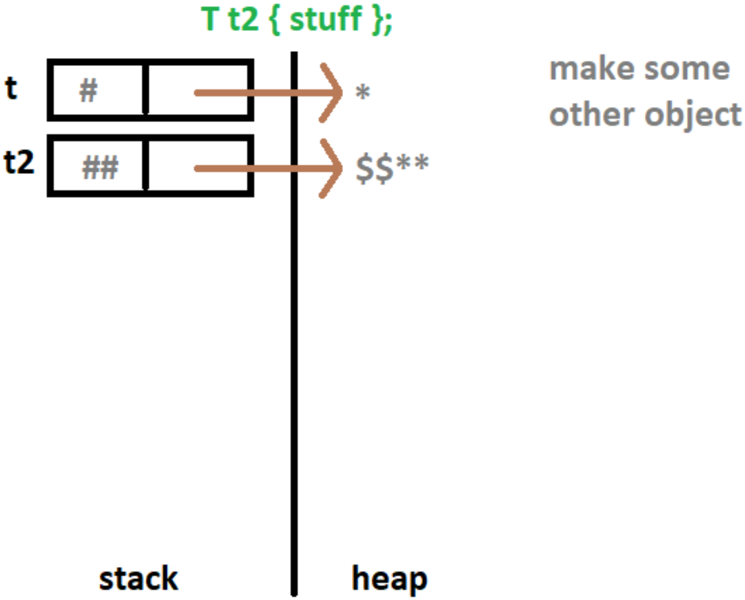
The depictions of the different constructors and the destructor illustrate what should happen for a well-programmed class, i.e. no memory leaks, etc.

Constructors and Destructors

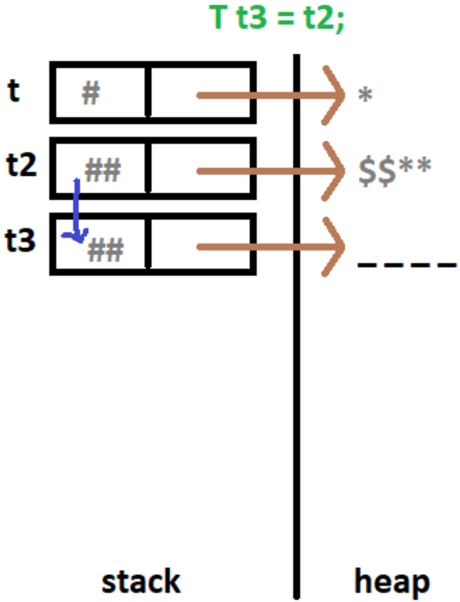


default
constructor
makes valid
object

Constructors and Destructors

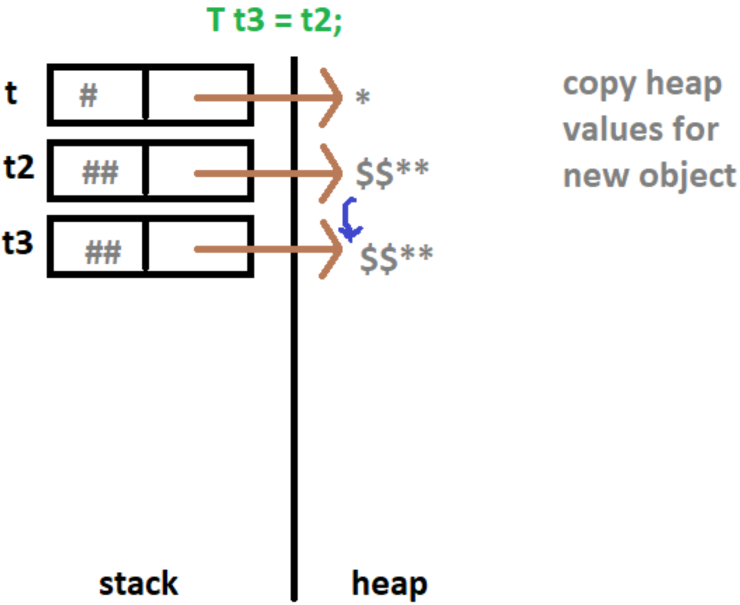


Constructors and Destructors

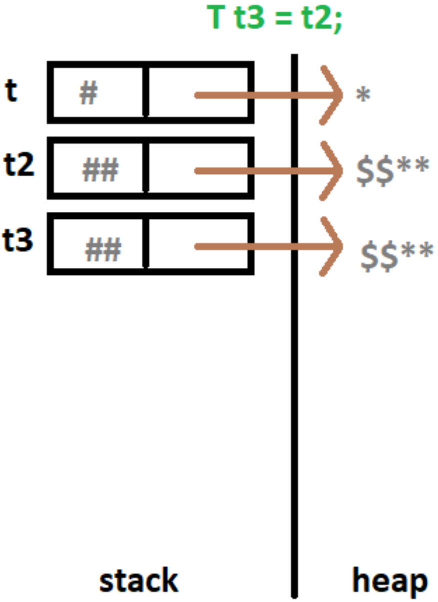


copy normal
stack
variables
and allocate
space on
heap for
deep copy

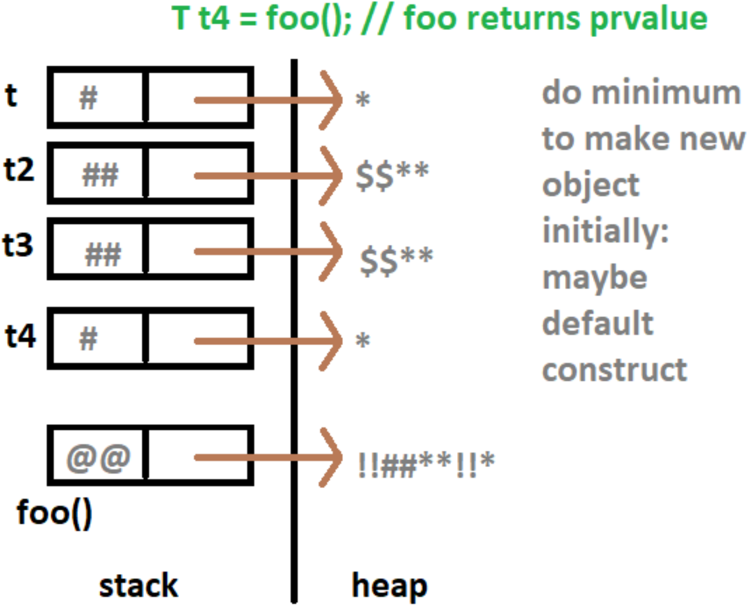
Constructors and Destructors



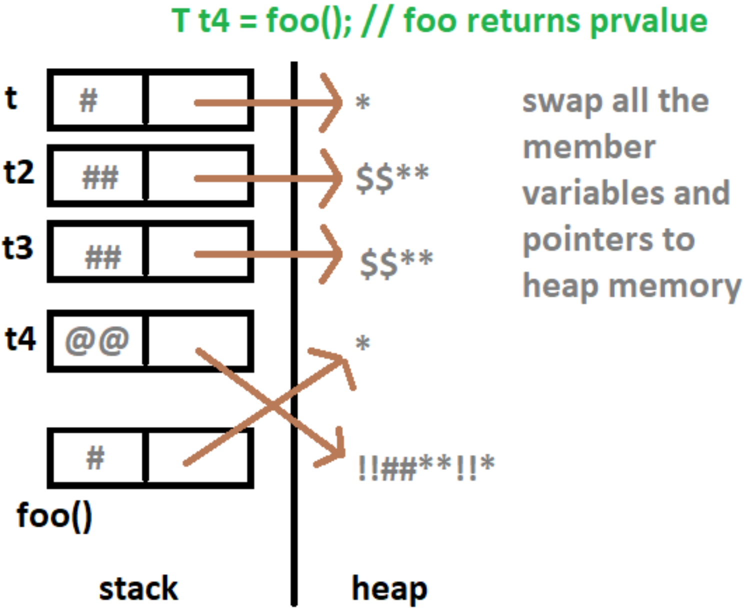
Constructors and Destructors



Constructors and Destructors

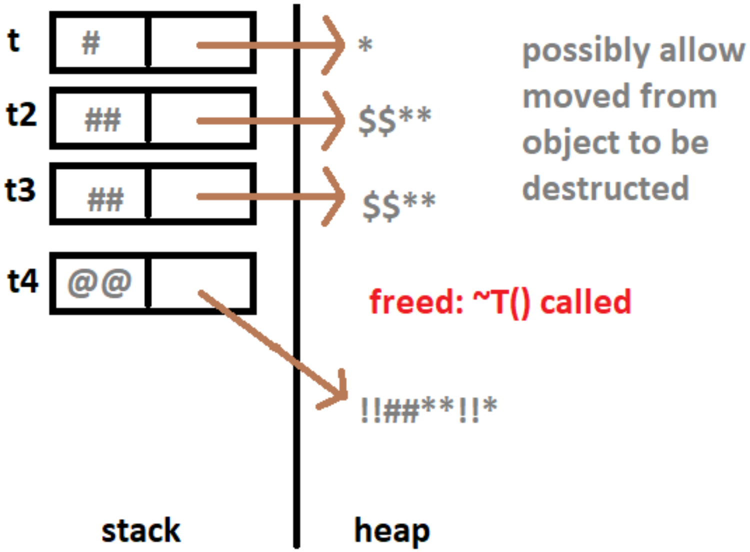


Constructors and Destructors

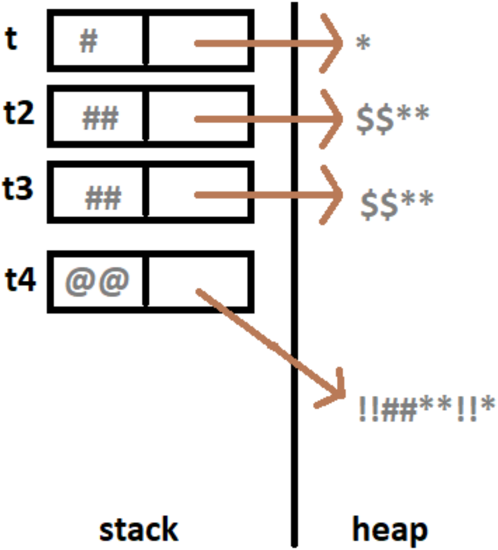


Constructors and Destructors

T t4 = foo(); // foo returns prvalue

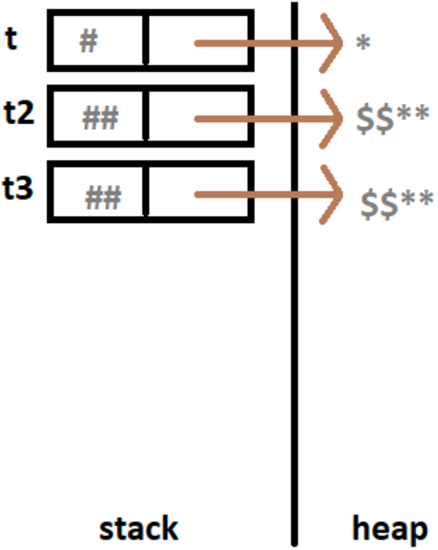


Constructors and Destructors



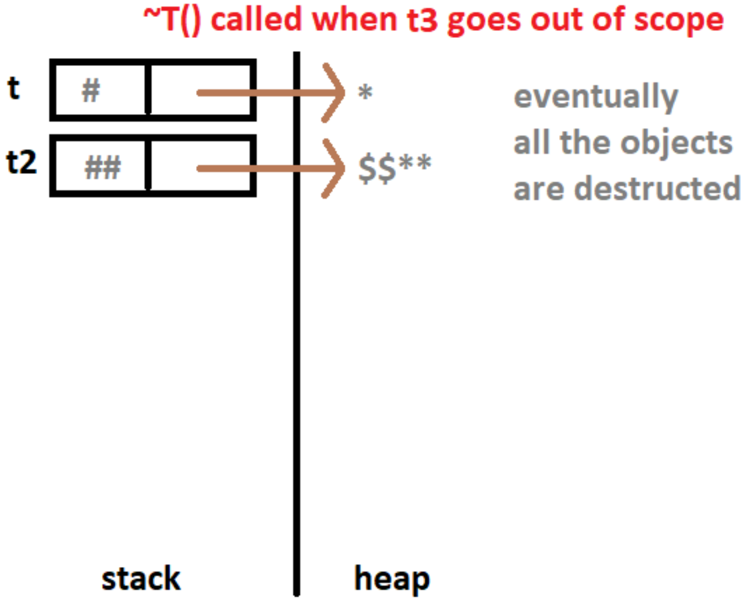
Constructors and Destructors

~T() called when t4 goes out of scope

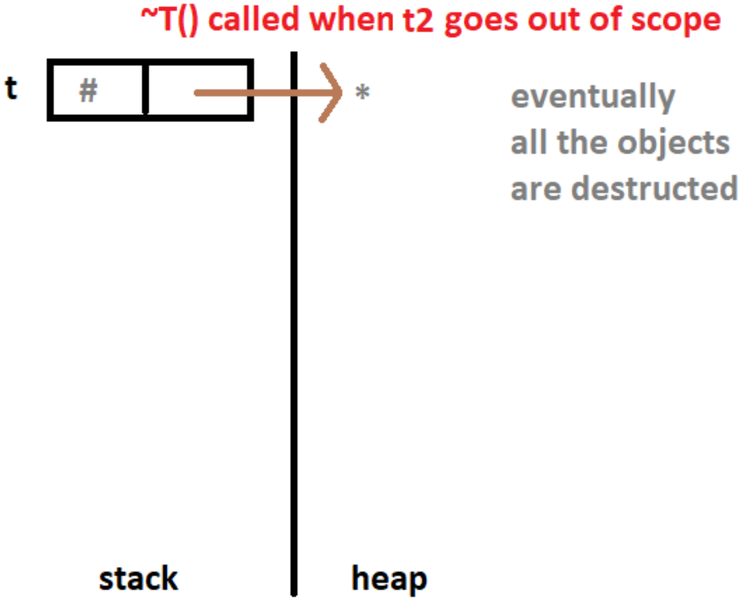


eventually
all the objects
are destructed

Constructors and Destructors



Constructors and Destructors



Constructors and Destructors

~T() called when t goes out of scope

eventually
all the objects
are destructed

stack

heap

Constructors and Destructors

stack

heap

A Simple String Class from Scratch

We will write our own simplified **string** class to explore these ideas. It will be defined within a namespace **basic**.

We will implement just a few operations to illustrate the concepts so far. We will be using raw pointers for this implementation.

A Simple String Class from Scratch I

```
namespace basic {
   class string {
   private:
      size_t sz; // the size of the object (counting null char)
      char *ptr; // points to heap memory of chars
```

A Simple String Class from Scratch II

```
  public:
    string();    // default constructor
    string( const char* );    // accept a string literal input
    string( const string& );    // copy constructor
    string ( string && );    // move constructor
    string& operator=(const string&); // copy assignment
    string& operator=(string&&); // move assignment
    ~string();    // a destructor
    void concat( const string& );    // a concatenation function
    void display() const; // print to screen
    char& at(size_t i); // at function for non-const strings
    char at(size_t i) const; // at function for const strings
  };
}
```

A Simple String Class from Scratch

Under certain conditions, the compiler will generate a move constructor for us, **moving** each member to construct the new class' members.

Likewise, under certain circumstances, the compiler may generate a copy constructor for us by **copying** each member to construct the new class (invoking the copy constructor for all class types, and directly copying primitive types like **int**, **char***, **double**, etc.). This is done as a **shallow copy** such as below:

```
string(const string& other) : sz(other.sz), ptr(other.ptr) {}
```

Often a **deep copy** that takes into account heap memory is preferred. Thus, we need to write our own copy/move constructors.

A Simple String Class from Scratch

For our implementations, we imagine that these definitions are given inside a set of braces specifying the **basic** namespace as **namespace basic { /* our definitions */ }**.

A Simple String Class from Scratch

For a default constructor, we implement

```
string::string() : sz(1), ptr ( nullptr ) {
   ptr = new char[1] {'\0'};
}
```

We initialize **sz** to 1 and make **ptr** point to a dynamic array storing only ('\0').

Generally strings are stored with a null terminating character to indicate the end of the string.

Remark: when dealing with *raw pointers that manage heap memory*, we often set them to **nullptr** in the constructor initializer list and then assign them dynamic memory in the body. More on this later.

A Simple String Class from Scratch

For the constructor accepting a string literal (**const char***):

```
// set size to 0 initially and point to null
string::string(const char* c) : sz(0), ptr(nullptr) {
    while (c[sz++] != '\0') // while not at null character, increment count
        { /* EMPTY BODY */ }

    // sz is now the size of the string literal, including the null character
    ptr = new char[sz]; // allocate large enough dynamic array

    for (size_t i = 0; i < sz; ++i) { // loop over string literal chars
        ptr[i] = c[i]; // c[i] same as *(c+i)
    }

}
```

A Simple String Class from Scratch

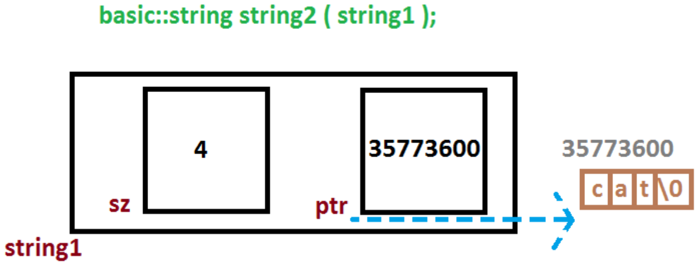
To generate the copy constructor, we construct a new dynamic array of appropriate size and copy the char values one-by-one.

```
// point to null and set size
string::string( const string& rhs)
    : sz( rhs.sz ), ptr ( nullptr ) {

    ptr = new char[rhs.sz]; // allocate new array

    // copy over all chars including the null
    for (size_t i=0; i < rhs.sz; ++i) {
        ptr[i] = rhs.ptr[i];
    }
}
```

Remark: we can access the private member variables of a **string** object with the dot (.) operator because we are defining a function of that class.

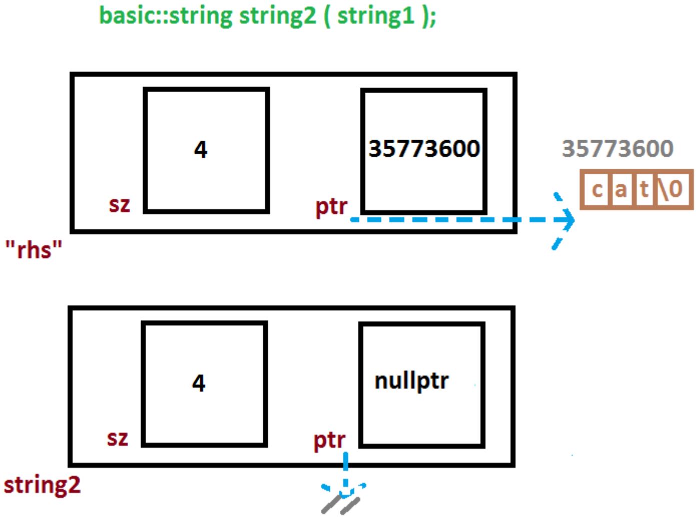
A Simple String Class from Scratch

Illustration of copy constructor:



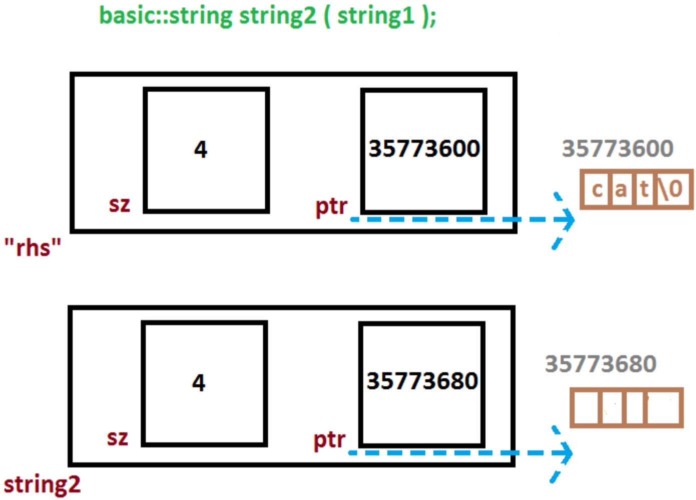
A Simple String Class from Scratch

Illustration of copy constructor:



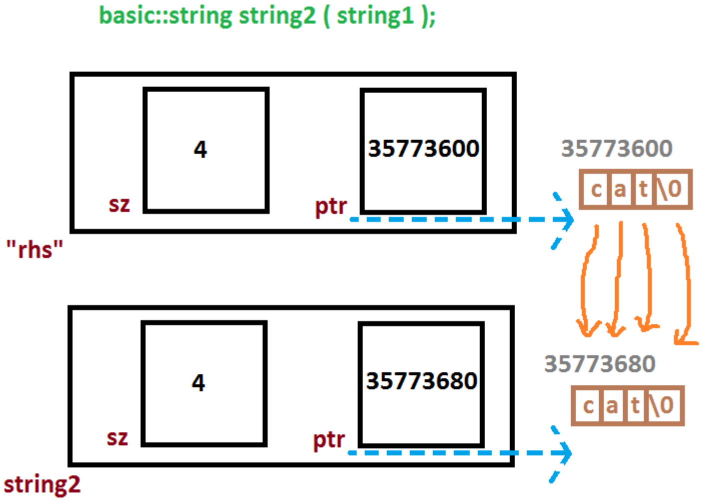
A Simple String Class from Scratch

Illustration of copy constructor:



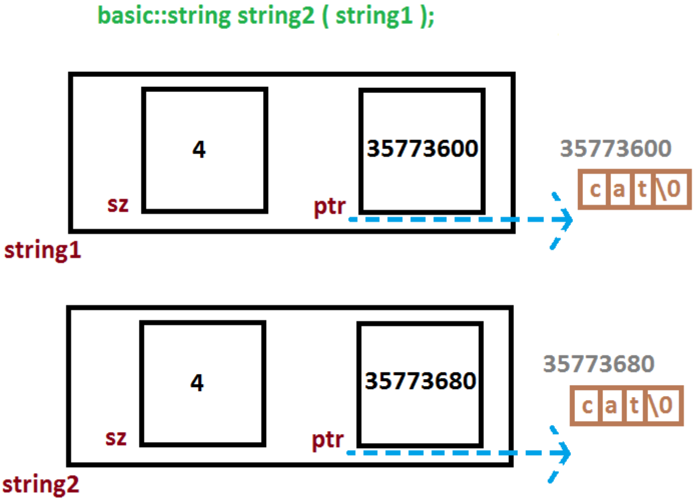
A Simple String Class from Scratch

Illustration of copy constructor:



A Simple String Class from Scratch

Illustration of copy constructor:



A Simple String Class from Scratch

The move constructor will take the resources from constructed-from object.

```
string::string ( string && rhs ) : string() { // invoke default constructor
   std::swap(ptr, rhs.ptr); // now switch resources
   std::swap(sz, rhs.sz); // and sizes
}
```

A Simple String Class from Scratch

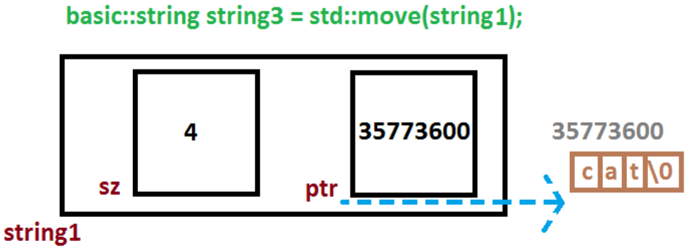
Within the constructor initializer list, one construtor can "delegate" its work to another constructor by giving the constructor name and arguments. We chose to default construct the **string** initially.

After it being default constructed, we swap the resources from the default object (storing only the null character) and the object we harvest from.

This has a nice property of making sure the "moved from" object is still in a valid state (it's just the empty string).

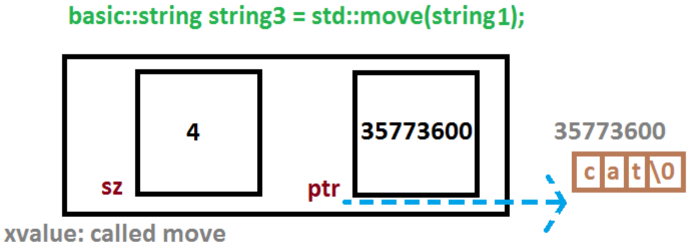
A Simple String Class from Scratch

Illustration of move constructor:



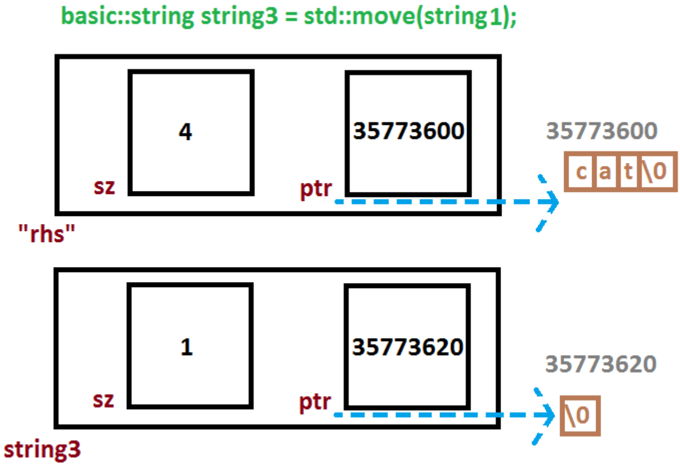
A Simple String Class from Scratch

Illustration of move constructor:



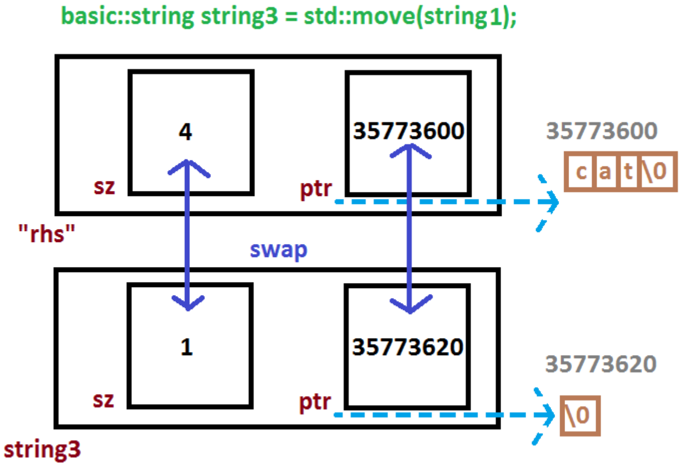
A Simple String Class from Scratch

Illustration of move constructor:



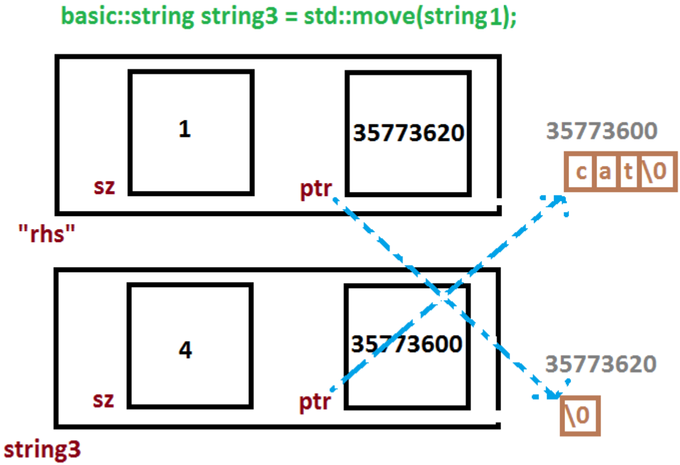
A Simple String Class from Scratch

Illustration of move constructor:



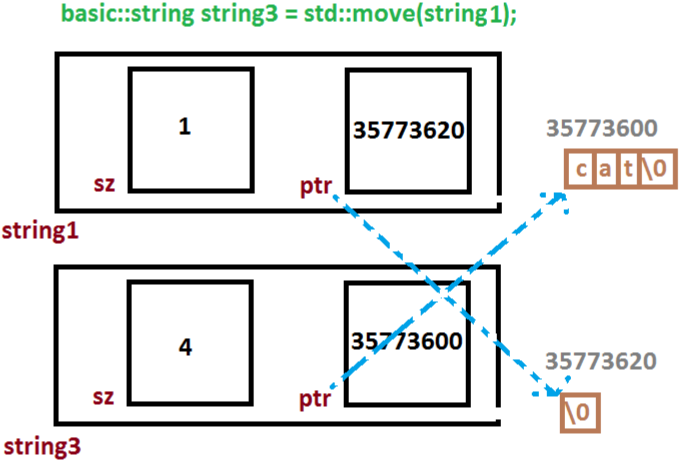
A Simple String Class from Scratch

Illustration of move constructor:



A Simple String Class from Scratch

Illustration of move constructor:



A Simple String Class from Scratch

Because we are not using smart pointers, we have to write our own destructor to free the heap memory. Fortunately, there isn't much work to do here.

```
string::~string() {
   // call the proper delete function for the dynamically allocated array

   delete [ ] ptr;
}
```

For debugging, sometimes it's nice to print a message in the destructor to know when it is being called.

A Simple String Class from Scratch

The **operator<<** has been overloaded for arguments **std::ostream&** and **const char*.** Thus, **std::cout** "already knows" how to print the characters of our string class assuming they are terminated by a null character.

To print the object:

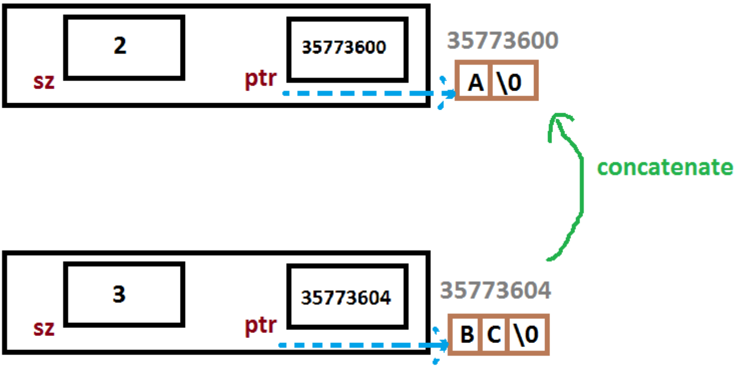
```
void string::display() const {
   std::cout <<ptr; // ptr is a char* (converted to const char*)
}
```

A Simple String Class from Scratch

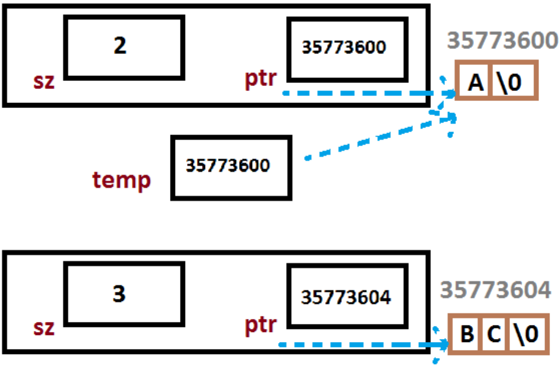
A longer approach, assuming we forgot about the preceding fact is:

```
void string::display() const {
   // loop over chars until at null character
   for (size_t i=0; ptr[i] != '\0'; ++i){
      std::cout <<ptr[i];
   }
}
```

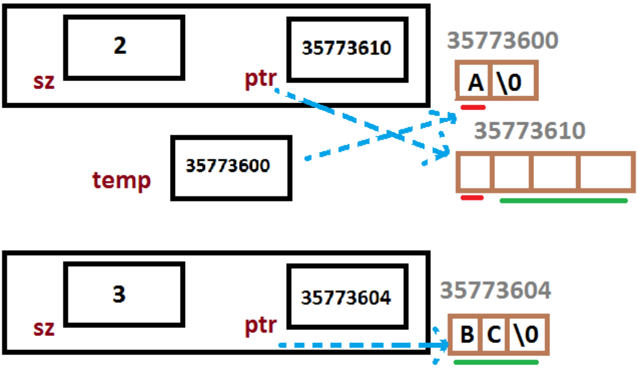
A Simple String Class from Scratch



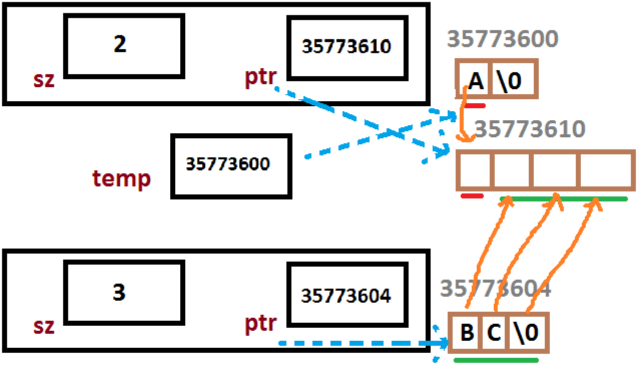
A Simple String Class from Scratch



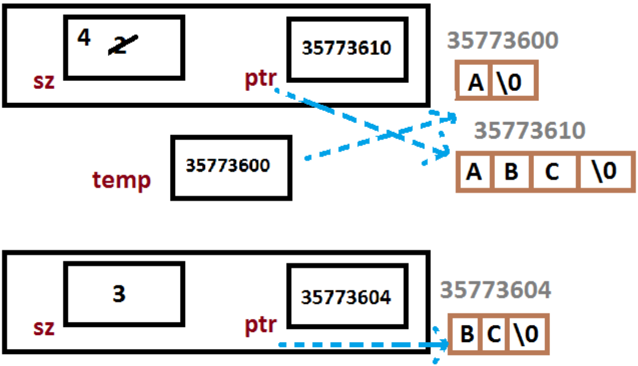
A Simple String Class from Scratch



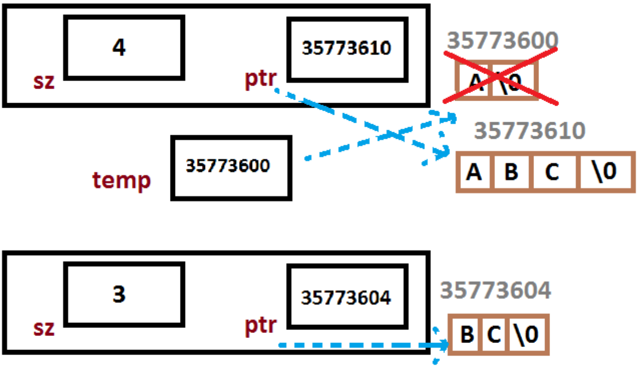
A Simple String Class from Scratch



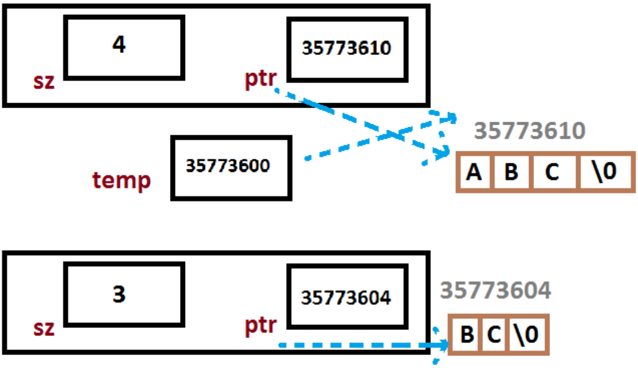
A Simple String Class from Scratch



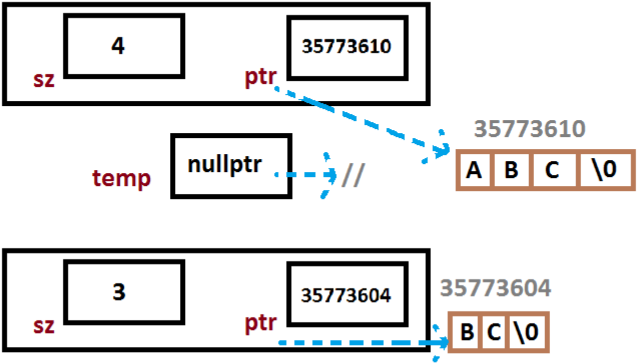
A Simple String Class from Scratch



A Simple String Class from Scratch



A Simple String Class from Scratch



A Simple String Class from Scratch

```
void string::concat(const string& rhs) {
   size_t newsize = sz + rhs.sz - 1; // don't need two null chars
   char* temp(ptr); // temporarily store pointer to the old elements
   ptr = new char[newsize]; // create new space

   // loop over but not include the null at sz-1
   for (size_t i = 0; i < sz - 1; ++i){
      ptr[i] = temp[i];
   }
   for (size_t i = 0; i < rhs.sz; ++i) { // copy second string data
      ptr[i + sz - 1] = rhs.ptr[i];
   }

   sz = newsize; // now update the size parameter
   delete [ ] temp; // free memory stored at the previous location!
   temp = nullptr;
}
```

A Simple String Class from Scratch

```
char &string::at(size_t i) { return ptr[i]; }

char string::at(size_t i) const { return ptr[i]; }
```

The **at** function, which just returns the **char** at a given index is said to be **overloaded on const.** If the object really is const, then only a copy of the **char** is returned, preventing mutations; but if the object is not const, a **char&** is returned, allowing for mutations.

```
basic::string s1("abc"); // not const
const basic::string s2("def"); // is const

s1.at(0) = ’A’; // okay, and s1 == "Abc"
char d = s2.at(0); // okay, d == ’d’

s2.at(0) = ’D’; // ERROR
```

A Simple String Class from Scratch

Many operators can be overloaded in C++. Here we specifically look at the **assignment operators**: the **copy assignment** and **move assignment** operators. They arise when we wish to overwrite the value of an already existing object.

```
basic::string x("X"); // x and y are constructed here
basic::string y("Y");

// copy assignment: x's old data will be gone and it will be a copy of y
x = y;

/* move assignment: y's old data will be gone and it takes the value of the
temporary value */
y = basic::string("Z");
```

As an **expression** such as **x=y** above, an assignment operator typically **returns the updated variable by reference**.

A Simple String Class from Scratch

Within every class, there is a special value, **this**, which is a pointer to the class itself.

And ***this** is therefore a reference to the class, possibly a reference to const if the object is const or an invoked member function is marked as const.

A Simple String Class from Scratch

```
struct Y {
   int a = 0;

   void print() const {
      std::cout <<a;
      // same thing: std::cout <<this->a; OR std::cout <<(*this).a;
   }
   void call_print() const {
      print();
      // same thing: this->print(); OR (*this).print();
   }

   // get marked const: *this is const Y&
   const Y& get() const { return *this; }

   // not marked as const: *this is Y&
   Y& get() { return *this; }
};
```

A Simple String Class from Scratch

Assignment operators must be implemented as member functions.

Similar to copy/move construction, we wish now to reassign the value of a **basic::string** when given either an lvalue or rvalue input.

A Simple String Class from Scratch I

```
string& string::operator=(const string& rhs) {
    if( this == & rhs) { // self-assign, do nothing
        return *this;
    }

    // effectively do the work of making a copy
    char *old(ptr); // store old place
    ptr = new char[rhs.sz]; // allocate new array
    for (size_t i=0; i < rhs.sz; ++i) { // copy over all chars including the null
        ptr[i] = rhs.ptr[i];
    }
```

A Simple String Class from Scratch II

```
    sz = rhs.sz; // update the size
    delete[] old; // free up old memory
    return *this;
}

string& string::operator=(string&& rhs) {
    // just swap the state of the two objects,
    // assigned-to object takes ownership of rhs.ptr memory
    std::swap(ptr, rhs.ptr);
    std::swap(sz, rhs.sz);
    return *this;
}
```

A Simple String Class from Scratch III

Remark 1: the check **this == &rhs** tests whether the address of the assigned-from and assigned-to objects are the same. If so, we avoid the work of extra copying.

Remark 2: some implementations (not ours!) actually delete the old memory first; in such cases, the self-assignment check is necessary to avoid destroying the object under self-assignment.

Remark 3: we will revisit assignment later to write a much slicker, single assignment operator that manages both assignment operators in fewer lines of code.

A Simple String Class from Scratch

The different constructors/assignments can arise in a variety of calls:

```
basic::string s0; // explicitly invokes default constructor
basic::string s1("aloha!"); // explicitly invokes string literal constructor
basic::string s2(s1); // explicitly invokes copy constructor

// explicitly invokes move constructor assuming foo() outputs rvalue
basic::string s3( foo() );
```

The above all *explicitly* invoke a constructor: there is no = sign in defining the objects.

A Simple String Class from Scratch

To *implicitly* invoke a constructor, the compiler "needs permission" to do so. The proper technical term is that the constructor is **non-explicit**.

```
/* implicitly invokes string literal constructor and requires a non-explicit
   copy/move constructor */
basic::string s4 = "UCLA";

basic::string s5 = s4; // implicitly invokes copy constructor
basic::string s6 = foo(); // implicitly invokes move constructor

/* may direct initialize s7 but requires a non-explicit copy or move
   constructor */
basic::string s7 = basic::string();
```

A Simple String Class from Scratch

s2 = s3; // copy assignment

s4 = foo(); // move assignment, assuming foo() outputs rvalue

Explicit Constructors

Constructors declared with the **explicit** keyword do not allow implicit calls.

```
struct Foo{
   Foo(int _i) : i(_i) {}
   int i;
};

// ...

Foo f{4}; // f.i == 4
Foo f2 = 5; // okay and f2.i == 5
```

Explicit Constructors

Constructors declared with the **explicit** keyword do not allow implicit calls.

```
struct Bar{
    explicit Bar(int _i) : i(_i) {}
    int i;
};

// ...

Bar b{4}; // b.i == 4
Bar b2 = 5; // ERROR: illegal call due to explicit constructor
```

Explicit Constructors

Remark: the constructors for a **std::shared_ptr** and **std::unique_ptr** accepting a raw pointer are explicit. That is why we cannot write:

```
// requires non-explicit constructor!
std::shared_ptr<int> sptr = new int;
```

Deleted Functions

Sometimes we wish to prevent a class from being copied, or block other functions upon the class. There is a **delete** keyword to allow for this. If we did not want the **class A** to be copied, the interface might look like:

```
class A {
public:
   A(const A&) = delete;
   // OTHER STUFF
};
```

Then we would not define **A(const A&)** anywhere (it has already been defined as deleted).

With the above, it would be impossible to construct an **A** by copying another lvalue. We usually do this when copying the class wouldn't make sense or would violate the intent of the class.

Deleted Functions

A **std::unique_ptr** has a deleted copy constructor. That is why we cannot copy them!

Stream objects, such as **std::cin** and **std::cout** which are **std::ostream** objects, must always be passed and returned by reference because the copy constructor is a deleted function and thus cannot be invoked.

Copy Elision

Due to compiler optimizations, sometimes the actual constructors invoked can be surprising... Consider the class below:

```
class X {
public:
   X() { } // default constructor
   X(const X &) { } // copy constructor
   // there is no move constructor!!!
};
```

and the function

```
X f() { return X{}; }
```

with the line of code (in main)

```
X x = f();
```

Copy Elision

We could wonder: how many times is the copy constructor invoked in the line

`X x = f();` ?

Strictly speaking:

- Within the body of **f()**, an **X** is constructed that is returned by value so a copy of the **X()** is returned.
- We need to construct an **X** object called **x** from that copy returned from **f()** and to do so, we need to copy **f()**'s output to make **x**.

Hence, there could be up to 2 calls to the copy constructor...

Copy Elision

But... the compiler may optimize to a direct construction; it may optimize to only make one copy; or it may make two copies. Modern compilers are supposed to optimize this sort of thing heavily, provided the constructors that it elides are accessible (not deleted, not explicit when invoked implicitly, etc.).

This eliding of the copy construtor is called **copy elision** and is part of C++ return value optimization.

Copy Elision

Because of the sometimes unpredictable choices the compiler makes, one should only use a constructor to create an object and to have no other side effects such as printing a message, etc.

This is also why, although technically not a direct construction, the code below could yield a direct construction anyway:

```
std::string s = std::string("pita");
```

Move Constructors for Classes with Class Member Variables

The function **std::move** converts its parameter to an r-value reference, suitable for efficient resource transfer. Most class objects of the standard library also have move constructors.

A move constructor with class object members should initialize those members from **moved**-from members of the class it harvests from.

Move Constructors for Classes with Class Member Variables

```
struct val_msg {
   int val;
   basic::string msg;

   val_msg() : val(), msg("heya") { } // val will be 0

   val_msg(val_msg&& right) : val(right.val), msg(std::move(right.msg)) { }

   // ...

};
```

Move Constructors for Classes with Class Member Variables

Consider the move constructor:

```
val_msg(val_msg&& right) : val(right.val), msg(std::move(right.msg)) { }
```

The **int** value can be taken directly since **int**s are fundamental types taking up little memory. But the **basic::string** should be transferred efficiently...

Move Constructors for Classes with Class Member Variables

```
val_msg(val_msg&& right) : val(right.val), msg(std::move(right.msg)) { }
```

Even though **right** references an rvalue (an object that will soon no longer exist and that we should harvest from), **right** itself is an lvalue (it has a name!). And **right.msg** is also an lvalue. If we just wrote

```
... msg( right.msg) ...
```

then to construct **msg**, we would invoke the copy constructor (**string(const string&)**) of **basic::string**! This is inefficient. But with

```
... msg(std::move(right.msg)) ...
```

msg is constructed via the **basic::string** move constructor (**string(string&&)**) and this is efficient.

Aside: Lvalue References in Operator Overloading

Remark: std::cout and the data types **std::ostringstream** and **std::ofstream** fall under the "umbrella" of the **std::ostream** data type.

This is why, for example in overloading **operator<<**, we could declare

```
std::ostream& operator<<(std::ostream&, const
   std::vector<std::string>&);
```

(and later define it) and such an operator would work with **std::cout**, or an output file stream object (**std::ofstream**), or an output string stream object (**std::ostringstream**) because an **std::ostream&** can bind to all of them when passed as an argument to the operator.

Similar rules apply for pointers.

Aside: Avoiding Exceptions with new Expression

When dealing with large amounts of data, it could be possible to run out of memory. In that case, the **new** expression can throw an exception.

By including the **<new>** header, we can detect when there is a memory allocation error because **new** will throw an exception of type **bad_alloc**. Handling of errors is a topic unto itself.

We can also request that **new** not throw an exception but instead return **nullptr** if the memory allocation fails.

```
int *x = new int; // may fail and throw an exception...
int *y = new(std::nothrow) int; /* if it fails then y==nullptr, otherwise y
   points to some heap int variable */
```

Aside: make_unique and make_shared and Safety

Prior to C++17, for robustness and exception safety, the use of **make_unique** and **make_shared** was preferred over the calls to the **new** expression in constructing smart pointers.

Aside: Try and Catch in Constructors

Later on after discussing exceptions, classes that manage heap memory through raw pointers may require more complicated looking constructors of the form

```
string::string() : ptr ( nullptr ), sz(1) {

    try { // try to allocate the memory
        ptr = new char[1] { '\0'};
    }
    catch(const std::exception& error) { // if an error
        delete [] ptr; // free up the memory
        ptr = nullptr; // make null
        throw; // throw exception again
    }
}
```

Summary

- References are bound to a block of memory; pointers, store addresses, can move
- Declaring something as const ensures (usually...) that it cannot be changed
- **new** allocates memory on the heap, returning a pointer
- **delete** frees the memory on the heap and destroys the object
- Smart pointers should be used to avoid memory leaks and other problems, but it can be dangerous to mix smart and raw pointers.
- The **shared_ptr** templated class allows for multiple references to an object; the **unique_ptr** only allows one.

Summary

- The **shared_ptr** templated class allows for multiple references to an object; the **unique_ptr** only allows one.
- All C++ expressions are: one of prvalue, xvalue, or lvalue.
- Lvalue references can bind to lvalues; rvalue references can bind to prvalues or xvalues.
- In managing heap memory directly, copy/move constructors, copy/move assignment operators, and destructors are essential for proper logic and efficiency.
- Identifying if an argument is an lvalue or rvalue is important for the sake of writing efficient code.
- Member functions can be overloaded on const.
- Compiler optimization can change the anticipated constructors, etc.

Exercises

1. What is the difference between...
   - a **pointer** and a **reference**?
   - an **lvalue** and an **rvalue**?
   - an **lvalue reference** and an **rvalue reference**?
   - **copy** constructors, **copy** assignment operators, **move** constructors, and **move** assignment operators (consider drawing a picture)?
2. Write an **Int** class that "wraps" around an **int.** It should store a single member variable **p** of type **int*** pointing to a value on the heap**.** Give
   - a default constructor, setting the **p** to **nullptr**;
   - a constructor accepting an **int**, making one on the heap and making **p** point to it;
   - a **set** function taking an **int** that changes the value **p** points to or, if **p** is null, makes **p** point to a new **int** with that value on the heap;
   - a **valid** function that returns **true** if the **p** is not null and **false** if it is null (this way one can check if **p** is null before potentially dereferencing it with **get** below);
   - a **get** function, overloaded on const, to retrieve the **int p** points to;
   - copy/move constructors;
   - copy/move assignment operators;
   - a destructor.

Exercises

3. Repeat the preceding exercise, making suitable modifications, with a smart pointer. You will not need to write a destructor.
4. How do **type** and **value** category differ?
5. Write a summary of how to identify **prvalues**, **xvalues**, **lvalues**, **rvalues**, and **glvalues** in expressions.
6. What are **memory leaks** and how can they be avoided?