61. Constructors

Objectives

- Understand what a constructor is
- Write constructors
- Write constructors using the initializer list
- Write inline methods
- Write overloaded constructors
- Understand how the default constructor works
- Write constructors with default values
- Write and use copy constructors

Constructor

Its always a good idea to initialize variables . So C++ encourages object initialization by giving a standardized way of initializing objects which is much better than the `init()` method in the `Date` class .

This is called the **class constructor** .

Techno-jargon: When you create an object `obj` from a class `C`, besides saying you are declaring `obj` from `C`, you would also say that you are **instantiating** `obj` from class `C`.

The constructor is just a method (i.e. member function.) Therefore, if you can write methods, you can write a constructor. However, there are two rules that you must follow :

1. The constructor has the same name as the class
2. The constructor does not have a return type (not even `void`)

Let's add a constructor to the `Date` class and remove the `init()` method.

First we replace the `init()` prototype in the class definition in the header file with the following constructor prototype:

```
// Date.h

class Date
{
public:
    Date(int, int, int);
    ...
private:
    int yyyy_, mm_, dd_;
};
```

Next, we change the name of the `Date::init()` method to `Date::Date()` in the implementation file (I'm also adding a print statement just for testing):

```
// Date.cpp

...

Date::Date(int yyyy, int mm, int dd)
{
    std::cout << "in Date::Date\n"; // for testing
    yyyy_ = yyyy;
    mm_ = mm;
    dd_ = dd;
}

...
```

Now run this and see what happens:

```
// main.cpp

...

int main()
{
    Date today = Date(2014, 12, 25);
    today.print();
    Date yesterday(2014, 12, 24);
    yesterday.print();

    Date * p = new Date(1970, 1, 1);
    p->print();
    delete p;

    return 0;
}
```

Note how I call the constructor ...

There are two different ways of calling the constructor in C++:

```
Date d(2014, 1, 1);
```

and

```
Date d = Date(2014, 1, 1);
```

They do the same thing.

And in the case of memory allocation of a Date object in the free store using new:

```
Date * p = new Date(1970, 1, 1);
```

There's a warning in using this version of calling the constructor:

```
Date d(2014, 1, 1);
```

It's better to use

```
Date d = Date(2014, 1, 1);
```

See the later section on Default Constructor.

Of course you can also construct a `Date` *value* without giving it to a variable name:

```
Date(2014, 12, 25).print();
```

In this case, you create a `Date` object (without a name) and use it to invoke `print`.

Exercise. Write a class `Being` with instance variables `num_heads`, `num_arms`, `num_legs`. Write a constructor to initialize all members. Write a `print` method in the class to print the values of all instance variables. Write getter and setter methods to get and set all instance variables. Test your class.

Exercise. Write a class `Vehicle` with instance variables `numWheels`, `numSeats`, `mileage (double)`, `year`, `make` (C-string, max size of 50), and `model` (C-string, max size of 50). Write a `print` method in the class to print the values of all instance variables. Write a constructor to initialize all members. Test your class.

Initializer List

Recall that in C++, arrays and `struct` variables can be initialized using an initializer list:

```
struct Z
{
    int x;
    double y;
};

Z z = {42, 3.14159}; // struct initialization
int x[] = {2, 3, 5, 9, 11}; // array initialization
```

Similarly, the member variables (instance variables) of an object can be initialized by an **initializer list** in the constructor.

Example:

```
// Date.cpp

...

Date::Date(int yyyy, int mm, int dd)
    : yyyy_(yyyy), mm_(mm), dd_(dd)
{}

...
```

In the above "`yyyy_(yyyy)`" means "`yyyy_` is initialized with `yyyy`".

Constructor initializers are important and unfortunately not always emphasized in books! (**Some books don't even mention them!!!**)

In a constructor, whenever possible, you should use initialization and not assignment like this:

```
// Date.cpp

...

Date::Date(int yyyy, int dd, int mm)
    : mm_(mm), dd_(dd) // Initialization ... GOOD!
{
    yyyy_ = yyyy;      // Assignment ... BAD!
}

...
```

Warning: If you do this:

```
// Date.cpp

...
```

```
Date::Date(int yyyy, int mm, int dd)
{
    yyyy_ = yyyy;
    mm_ = mm;
    dd_ = dd;
}

...
```

you should know by now that you're using assignment and not initialization.

This means that `yyyy_` was

1. initialized to random garbage and *then*
2. assigned the value of `yyyy`.

Therefore, because of the above (*and* also other reasons) using initializers will always be faster. Do not let me ever catch you doing an assignment in constructors when it should be an initialization using an initializer list.

In fact there are cases where you **have** to use initializers:

```
// C.h

class C
{
public:
    C(int &);
private:
    int & x_;
    const double y_;
};
```

```
// C.cpp

C::C(int & x)
    : x_(x), y_(3.14159)
{}
```

Why? Because references and constants **must be initialized**.

The following is a quick review of the relevant facts. Check notes on constants and references for details.

```
const int x; // WRONG: constants MUST be initialized!
x = 42;      // TOO LATE!!!

int j = 42;
int & k;    // WRONG: References must be initialized!
k = j;      // TOO LATE!!!
```

It should have been this:

```
const int x = 42; // THAT'S BETTER!!!
```

```
int j = 42;
int & k = j;      // YES!!!
```

By the way, you can use any valid expression to initialize the object's members and the body of the method can still have statements:

```
// C.h

class C
{
public:
    C(int, int);
private:
    int x_, y_, z_;
};
```

```
// C.cpp

C::C(int a, int b)
    : x_(a + b), y_(a * b - 1), z_((x_ + y_) * a * b)
{
    x_ = 42;
}
```

Exercise. Rewrite the constructor of the `Being` class so that the constructor uses an initializer list.

Exercise. Rewrite the constructor of the `Vehicle` class so that the constructor uses an initializer list. (Note that for the string members, you have to copy the characters of the strings in the body of the constructor. So the body is not empty.)

Gotchas

Here are two very common gotchas for you …

First: The initializer syntax can only be used in constructors:

```
// Date.cpp
...
void Date::set_d(int dd)
    : dd_(dd)        // BAD BAD BAD WRONG WRONG WRONG
{}
```

Second: Order of initialization is the order of member declaration:

```
// C.h
class C
{
public:
    C(int, int);
private:
    int avg_, x_, y_;
    // avg_ = average of x_, y_
};
```

```
// C.cpp

C::C(int a, int b)
    : x_(a), y_(b), avg_((x + y) / 2)
// avg_ is initialized with x_, y_ before they are
// initialized!!! YIKES!!!
{}
```

To fix this problem, either
(1) use `a` and `b` in the initialization of `avg_`:

```
// C.cpp

C::C(int a, int b)
    : x_(a), y_(b), avg_((a + b) / 2)
// avg_ doesn't rely on x_ and y_, so it's safe
{}
```

or
(2) declare `avg_` after `x_` and `y_` in the class definition (after all, the `avg_` average should clearly be set after `x_` and `y_`, right?!?!)

```
// C.h

class C
{
public:
    C(int, int);

private:
    int x_, y_, avg_;
    // avg_ = average of x_, y_
};
```

Review: Inline Functions

Review the notes on inline functions. The following is only a quick recap.

Recall that an inline function is like a rubber stamp for a function:

```
#include <iostream>

inline int max(int x, int y)
{
    return (x < y ? y : x);
}

int main()
{
    std::cout << max(2, 3) << '\n';

    return 0;
}
```

This means that there is really no `max()` function. Rather, at the place where the inline function is called, C++ substitutes the code for `max()` so that the program compiled is:

```
#include <iostream>

int main()
{
    std::cout << (2 < 3 ? 3 : 2) << '\n';

    return 0;
}
```

Inline functions can speed up program execution since the act of making a function call takes time.

The compiler can choose to ignore making a function an inline function so that it becomes a real function. This can happen if the function is too long.

For multi-file compilation, the *whole* inline function must be in the header file. You cannot have an inline prototype in the header file and the definition of the body of the function in the implementation file.

Inline Methods

The information in this section applies to inlining of methods and not just constructors.

Class methods (i.e., member functions) can also be inlined.

To inline a class method, you can either:

1. Put the whole method inside the class definition.
2. Put the method definition in the header file prepended with `inline`.

First way:

```
// Date.h

class Date
{
public:
    Date(int yyyy, int mm, int dd)
        : yyyy_(yyyy), mm_(mm), dd_(dd)
    {}

    ...
};
```

```
// Date.cpp

// Date::Date(int, int, int) moved to Date.h

...
```

Second way:

```
// Date.h

class Date
{
public:
    Date(int yyyy, int mm, int dd);
    ...
};

inline Date::Date(int yyyy, int mm, int dd)
    : yyyy_(yyyy), mm_(mm), dd_(dd)
{}
```

```
// Date.cpp

// Date::Date(int, int, int) moved to Date.h

...
```

Exercise. All the methods in our `Date` class are pretty short. Inline them.

Exercise. Inline all the short methods in the `Being` class.

Exercise. Inline all the short methods in the `Vehicle` class.

Review: Function Signature

Review notes on function loading. This is a quick recap.

The **signature** of a function is the list of types of the parameters of the function.

WARNING: Note that the return type is ***not*** part of the function signature.

Here's an example: the signature of

```
int f(int x, char y) {}
```

is `(int, char)`

Here are some warnings:

- The type of “`const X`” is `X`.
- The type of “`const X *`” is `const X *`.
- The type of “`X[]`” is `X*`.

Here’s a trick to get your C++ compiler to tell you the signature of a function. Deliberately include an error in the function.

```
// main.cpp

void f(int x, const int y, const int * const z)
{
    abc;
}

int main()
{
    f();

    return 0;
}
```

When you compile, the compiler will have to tell you there’s an error in the above function, listing the prototype. For g++, you will get:

```
main.cpp: In function void f(int, int, const int*):
main.cpp:3:5: error: abc was not declared in this
scope
```

You can see from the above error message that for the second parameter, although it's `const int`, the type as part of the signature is `int`.

Exercise. Pair up functions below with the same signatures. Find a way to check your answer with your compiler.

```
void f(int x, const int & y) {}
void f(int x, const int * y) {}
void f(int x, const int y) {}
void f(char x, const int y) {}
```

```
void f(const int y, char x) {}
void f(int * x) {}
void f(int x[]) {}
void f(int x[10]) {}
void f(int x[10][10]) {}
void f(int x[10][]) {}
```

Review: Function Overloading

A function is **overloaded** if its name is used more than once.

The signature of the functions with the same name must be different. That's because in C++, a function is identified not by its name, but by it name and its signature, i.e., the name and the list of types of its parameters.

Exercise. Does this compile? Next, check with your compiler.

```
void f(int x){}            // f and signature int
void f(int y, int y){}     // f and signature (int,int)
void f(int x, double y){} // f and signature (int,double)


int main()
{
    return 0;
}
```

Exercise. Does this compile? Next, check with your compiler.

```
void f(int x) {}
int f(int x) {}
void f(int x, double y) {}

int main()
{
    return 0;
}
```

Exercise. Does this compile? Next, check with your compiler.

```
void f(int * x) {}
int f(int x[]) {}

int main()
{
    return 0;
}
```

When a function is called in your code, C++ will use the function that matches both the function name and the type of the values passed in with the candidate’s signature. If none is found, C++ will try to typecast the arguments. The one used is the one with the least number of typecasts. If there isn’t one, you get an ambiguous invocation error.

Exercise. Which function is called (or is there an error)?

```
void f(int x) {}
```

```
void f(double x) {}

int main()
{
    f(1);
}
```

Check by running this version:

```
#include <iostream>

void f(int x)
{
    std::cout << "f(int)\n";
}

void f(double x)
{
    std::cout << "f(double)\n";
}

...
```

Exercise. Which function is called (or is there an error)?

```
Void f(double x, int y) {}
void f(int x, int y) {}
void f(int x, double y) {}
void f(double x, double y) {}

int main()
{
    f(1, 1);

    return 0;
}
```

Check with a program.

If you compile the following with g++:

```
void f(double x, int y) {}
void f(int x, double y) {}
int main() { f(1, 1); }
```

it will complain with this message:

```
a.cpp: In function int main():
a.cpp:3:20: error: call of overloaded f(int, int) is
ambiguous
a.cpp:1:6: note: candidates are: void f(double, int)
a.cpp:2:6: note: void f(int, double)
```

Basically, when looking for a function to execute

```
f(1, 1);
```

there is no exact match. So C++ attempts to typecast. If C++ typecasts the first value to 1.0:

```
f(double(1), 1);
```

it will be able to match this:

```
void f(double x, int y) {}
```

and if C++ typecasts the second value to get 1.0:

```
f(1, double(1));
```

it will match this:

```
void f(int x, double y) {}
```

So, in this case, since both functions match `f(1, 1)` with 1 typecast, your C++ compiler does not know what to do and will complain. Of course if there's no exact match and there's only one option with one typecast, then C++ will happily choose that function that matches `f(1, 1)` with one typecast.

Method overloading

Methods in a class can also be overloaded as long as the signatures are different.

```
class C
{
public:
    void f(int);
    void f(int, int);
    void f(int, double);
};
```

In this case, all the following have different signatures.

```
C::f(int)
C::f(int, int)
C::f(int, double)
```

Note that since every method is within a class scope, you can have methods with the same name and same signatures but in different classes. So the following is OK:

```
C::f(int x) {}
D::f(int x) {}
```

since if an object `obj` is of class `C`, then `obj.f(42)` will result in `C::f` being executed. There's no ambiguity.

We will be overloading the `Date` constructor: We will later write a "default" constructor and a "copy constructor".

I'll write another constructor so that I can do something like this:

```
Date date1("December 25, 1985");
```

The prototype is of course like this:

```
Date(char s[]);
```

The pseudocode should be like this:

```
read enough character from s to compute the value for mm_
read past a space
read till a comma is reached, computing the value for dd_
read past a space
read remaining characters to compute the value for yyyy_
```

And we get:

```
Date(char s[])
{
    int i = 0;
    if (s[0] == 'J' && s[1] == 'a') mm_ = 1;
    else if (s[0] == 'F' && s[1] == 'e') mm_ = 2;

    ...
```

```
    i = 2;
    while (s[i] != ' ') i++;
    i++;

    dd_ = 0;
    while ((0 <= s[i] - '0') && (s[i] - '0' <= 9))
    {
        dd_ *= 10; dd_ += (s[i] - '0'); i++;
    }

    while ((s[i] < '0') || (s[i] > '9')) i++;

    yyyy_ = 0;
    while ((0 <= s[i] - '0') && (s[i] - '0' <= 9))
    {
        yyyy_ *= 10;
        yyyy_ += (s[i] - '0');
        i++;
    }
}
```

Default constructor

The **default constructor** of a class is the constructor that does not have any parameters.

Why use a default constructor? Because they make it easy to construct the most common objects.

Let's add one more constructor (i.e. we are overloading the constructor) to our Date class.

```
// Date.h
...

class Date
{
public:
    Date(int, int, int);
    Date();
    ...
};

...
```

```
// Date.cpp
...

Date::Date()
    : yyyy_(1970), mm_(1), dd_(1)
{}

...
```

How do you call the default constructor? Easy.

If `C` is a class, the following are two different ways to call the default constructor:

```
C obj1;
C obj2 = C();
```

Try this with our `Date` class:

```
Date date1;              // date1 calls Date::Date()
date1.print();
Date date2 = Date();     // date2 calls Date::Date()
date2.print();
Date * date3 = new Date(); // Date::Date() called
date3->print();
```

To verify that we did go into the default constructor, run the program again with a print statement in the default constructor.

Here's an important **WARNING** … From this syntax of calling constructors:

```
Date today(2014, 12, 25);
```

you might be tempted to do this:

```
Date date1(); // trying to call Date::Date() ...
              // WRONG!!!
```

This is a subtle problem as the compiler will not complain and in fact you can even run the program.

The problem begins when you use `date1`:

```
Date date1();  // Date::Date() ... WRONG!
date1.print(); // Compiler now complains ... :(
```

Why is that?!? Because when you write this:

```
Date date1();
```

Your compiler thinks that you're writing down a function prototype!!! In other words, your compilers thinks that `date1` is a function with no parameters and returns a `Date` value.

Summary:

```
Date date1;          // Date::Date() OK
Date date2 = Date(); // Date::Date() OK
Date date3();        // Date::Date() ... WRONG!!!
```

The FREE default constructor

For any class, C, if you do not specify any constructor, C++ will automatically add the default constructor that does not do anything into the class:

```
class C
{
public:
    C() {}
};
```

In fact, that's the reason why our first Date class works. Recall that for that version (without explicitly defining constructors), I was able to do this:

```
...
int main()
{
    Date today;
    today.init(2014, 12, 25);
    ...
    return 0;
}
```

Of course now you know that in fact:

```
...
int main()
{
    Date today; // default constructor Date::Date() was
                // called to initialize the members of
                // today. In particular today.yyyy_,
                // today.mm_ and today.dd_ was initialized
                // to random values.
    today.init(2014, 12, 25);
    ...
    return 0;
}
```

This means that when using the default constructor, the object being initialized will have random initial values.

If you write one or more constructors in this class, C++ will **not** add this do-nothing default constructor into this class. This is very important because later you will see that there are times when you need to have a default constructor, whether it is supplied by your C++ compiler (automatically) or by you (manually).

Exercise. Create a default constructor for the `Being` class with the following default values: 2 for number of heads, 3 for number of arms, and 2 for number of legs. Test your code.

Exercise. Create a default constructor for the `Vehicle` class with the following default values: 4 for number of wheels, 4 for number of seats, 45.24 for mileage, 2011 for year, “Nissan” for make, and “Sentra” for model. Test your code.

Review: Default values for functions

Review the notes on default values for functions. This is only a quick recap.

Recall: Parameters of a function can have default values.

If a default value is assigned to a parameter, then all parameters to the right must also have default values:

```
void f(int x, int y = 0, int z = 0); // OK
void f(int x, int y = 0, int z);     // BAD!!!
```

You can then invoke the function without specifying values for default-valued parameters. If you do not specify a value for such a parameter, the default value is used:

```
void f(int x, int y = 0, int z = 0) { ... }

int main()
{
    f(42, 24); // i.e., in f, x = 42, y = 24, z = 0
}
```

For multi-file compilation, default values are in the header file, not in the cpp file:

```
// xyz.h

#ifndef XYZ_H
#define XYZ_H

int f(int=0, int=42);

#endif
```

```
// xyz.cpp

#include "xyz.h"

int f(int x, int y)
{
    return x + y;
}
int main()
{
    return 0;
}
```

Exercise. Does this program compile?

```
void f(int x) {}
void f(int x, int y = 42) {}
int main() {}
```

Exercise. Which function is called (or is there an error)?

```
void f(int x) {}
void f(int x, int y = 42) {}
int main()
{
    f(1);
    return 0;
}
```

Default value for methods

The above information on default values for functions also applies to methods of classes.

Modify our `Date` class:

```
class Date
{
public:
    Date(int = 1970, int = 1, int = 1);
    // remove default constructor ... why?
    void set_d(int = 1);
    void set_m(int = 1);
    void set_y(int = 1970);
    ...
};
```

Now modify the implementation file `Date.cpp`. Add test cases to `main()`.

Exercise. For the `Being` class, rewrite the constructor so that it has default values using the values from the default constructor. Modify the setter methods to use the corresponding default values.

Exercise. For the `Vehicle` class, rewrite the constructor so that it has default values using the values from the default constructor. Modify the setter methods to use the corresponding default values.

Copy constructor

Suppose you already have a `Date` object `date1`. You want to create another `Date` object, `date2`, so that `date2` has the same values as `date1`.

`date2` is not a reference to date1 like this:

```
Date & date2 = date1;
```

Rather, you want `date2` to be an actual Date object with it's own member variables.

You can do this

```
Date date2(date1.get_y(), date1.get_m(), date1.get_d());
```

using the constructor that accepts three integers. But ...

What a pain!

This is cleaner:

```
Date date1;
Date date2;
date2 = date1;
```

But this means that `date1` was first initialized to something that I do not want, and *then* I assign the values of `date1` to `date2`. That's not efficient.

We can achieve a cleaner initialization if we define a constructor that accepts a `Date` object. In other words I would like to do this:

```
Date date2(date1);
```

The prototype looks like this:

```
// Date.h

...

class Date
{
public:
    Date(const Date &);

    ...

};

...
```

Note that I'm passing by reference to make the parameter passing efficient and since I don't intend to modify the parameter, I make it a constant. I'm therefore passing by constant reference.

The implementation looks like this:

```
// Date.cpp
Date::Date(const Date & date)
    : yyyy_(date.yyyy_), mm_(date.mm_), dd_(date.dd_)
{}
```

You can think of `date2` as a clone of `date1`. You can (and should) think of the copy constructor as a cloning operation.

For any class `C`, the **copy constructor** of that class is a constructor with the following prototype:

```
C(const C &);
```

Think of the copy constructor as a **cloning operation** and the intent is to copy values from one object to another.

Exercise. Inline our `Date` copy constructor.

Calling the copy constructor

There are actually two different syntax for invoking the copy constructor of a class `C`:

```
// obj1 is an object of class C

C obj2(obj1); // clone obj2 as obj1
C obj3 = obj1; // clone obj3 as obj1
```

WARNING: The second syntax is calling the copy constructor, not the assignment operator.

Example. Insert a print statement in the copy constructor of `Date`:

```
Date(const Date & date)
    : yyyy_(date.yyyy_), mm_(date.mm_), dd_(date.dd_)
{
    std::cout << "Date::Date(const Date &)\n";
}
```

and then execute this in `main()`:

```
Date date1(2050, 1, 1);

Date date2(date1);  // clone date2 from date1
Date date3 = date1; // clone date3 from date1
```

Default copy constructor

If a copy constructor is not specified, your C++ compiler will supply a default copy constructor that initializes all instance variables of the object invoking the constructor with the corresponding members of the object that is passed in.

```
// C has instance variables a_, b_, c_, d_
C::C(const C & obj)
    : a_(obj.a_), b_(obj.b_), c_(obj.c_), d_(obj.d_)
{}
```

So technically we don't need the copy constructor for our `Date` class. However, you will see very soon, that there are cases (many cases!!!) where you want your own copy constructor to do something else. So here's the warning just so you catch it ...

WARNING: There are cases where the default member-by-member copy is ***not*** what you want. (See later examples.)

If you define a copy constructor, of course the compiler will not supply one. You can't possibly have two copy constructors. (Why?)

Comment out our copy constructor in `Date` and then execute in `main()`:

```
Date date1(2050, 1, 1);
Date date2(date1);  // clone date2 from date1
Date date3 = date1; // clone date3 from date1
```

You will find that the compiler will not complain and the program does run. Why? Because your C++ compiler supplies the default copy constructor.

Exercise. Write a copy constructor for the `Being` class.

Exercise. Write a copy constructor for the `Vehicle` class.

Exercise. Here's the header file for weather control devices:

```
// WeatherCtrl.h
class WeatherCtrl
{
public:
    WeatherCtrl(double, double);
    double get_temp();
    double get_pressure();
    void set_temp(double);
    void set_pressure(double);

private:
    double temp_;
```

```
    double pressure_;
};
```

Write a cpp file containing the method implementation (i.e., definition) for all the prototypes that appear in the header file.

Next go over the following `main.cpp` that uses the weather control class:

```
// main.cpp
int main()
{
// Declare and initialize wc as a WeatherCtrl object with
// initial temperature reading of 50.5 and pressure reading
// of 30.5

// Print the temp and pressure value of wc using std::cout

// Set the temp and pressure value of wc to 60 and 40
// print the temp and pressure value of wc using std::cout

// Uncomment the next statement and compile. What's wrong?
// WeatherCtrl wc2;

// Comment out the above wrong statement and
// move on ...

// Modify the constructor so that it's inlined in
// the header and defaults the temp and pressure
// value to 50 and 60. Use an initializer list.

// Uncomment the next line, compile and run.
// WeatherCtrl wc2;

// Print the temp and pressure value of wc2.

// Declare wc3 as a pointer to WeatherCtrl and
// initialize it to point to a WeatherCtrl object in the
// heap with 110 and 70 for temp and pressure
// values.
// Print the temp and pressure of the object wc3 is
// pointing to.

    return 0;
}
```

Exercise. If `C` is a class and `obj1` is an object of class `C` and you want to clone `obj1` to another object, say `obj2`. The following works but it's a bad idea. Why?

```
C obj2 = C(obj1);
```

If you don't believe me you can try

```
Date date1(1970, 1, 1);
Date date2 = Date(date1);
date2.print();
```

Exercise. The following `Int` class essentially models an integer. Complete it.

```
class Int
{
public:
    Int(int x)
        :              // initialize x_ with x
    {}

    Int(             ) // copy constructor
        :
    {}

    int get()
    {
        return x_;
    }

private:
    int x_;
};
```

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

int main()
{
    Int i(5);

    // Construct object j of type Int
    // using i. Use the copy constructor

    std::cout << j.get() << '\n';

    return 0;
}
```

Now write a set method to set the value of x in the object. Test it with this:

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

int main()
{
    Int i(5);
    // Construct object j of type Int
    // using i. Use the copy constructor.
    std::cout << j.get() << '\n';

    i.set(42);
    std::cout << i.get() << '\n'; // should be 42

    return 0;
}
```

Finally, add an increment method so that this works:

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

int main()
{
    Int i(5);
    // Construct object j of type Int
    // using i. Use the copy constructor.
    std::cout << j.get() << '\n';

    i.set(42);
    std::cout << i.get() << '\n'; // should be 42

    i.increment(j); // i.x is incremented by j.x
    std::cout << i.get() << '\n'; // should be 47

    return 0;
}
```

Example and exercise: `IntPointer`

The next few exercises are extremely important!!! Some will in fact appear in assignments!!!

One of the problems regarding pointers (pointing to either a single value or an array of values) is that there's a danger of forgetting to deallocate memory:

```
void f()
{
    int * p = new int;
    int * q = new int[10];
    // do something with *p
    // do something with q[0],...,q[9]
    return; // oops ... forgot delete p and
            // delete [] q;
}

int main()
{
    f();
    // oops ... memory leak ... lost 11 integers!!!
    return 0;
}
```

You will see later (see notes on Destructors) that classes will help solve this very dangerous problem. For this section, let's talk about a pointer to a single value and write a class for that.

The following class models an integer pointer. This is an important example. But it's incomplete. (Clearly once you're done with this, you can also talk about a class to model the pointer to a double, to a char, etc.)

```
class IntPointer
{
public:
    IntPointer(int x)
        : p_(new int) // allocate memory for p_
    {
        *p_ = x; // store an int at memory that p_
                // points to
    }

    IntPointer(const IntPointer & intptr)
        : p_(new int)
    {
        *p_ = *(intptr.p_);
    }

    int dereference()
    {
        return *p_;
    }

```

```
    void allocate()
    {
        if (p_ == NULL)
        {
            p_ = new int;
        }
    }

    void deallocate()
    {
        delete p_;
    }

private:
    int * p_;
};
```

Exercise. Here’s a program that uses the above class. Trace the program by hand. Draw a picture of the memory during execution. What is the output?

```
// What must you #include?

void f()
{
    IntPointer p(42);
    IntPointer q(p);
    std::cout << q.dereference() << '\n';
    // memory leak here
}

int main()
{
    f();

    return 0;
}
```

Check your trace by completing the above code and running it. Can you explain why there’s a memory leak?

Exercise. To make the above object more like a pointer, modify the `dereference` method:

```
int operator*() // change deference to this
{
    return *p_;
}
```

and in `f` change the call to `dereference()` to `*`:

```
void f()
{
    ...
    std::cout << *q << '\n'; // *q = q.operator*()
    ...
}
```

Run the program again.

Exercise. However this does not work:

```
void f()
{
    ...
    std::cout << *q << '\n';
    *q = 1024;                    // YIKES
    std::cout << *q << '\n';
    ...
}
```

Compile and read the error message - remember it! It seems that getting `*q` works but modifying `*q` does not work.

Execise. To correct it, change `operator*` to this:

```
int & operator*()
{
    return *p_;
}
```

Compile and run. Remember this fix!

Execise. Now add a line of code to deallocate memory so that there's no memory leak. It's still a problem that we have to deallocate memory before object `q` dies (i.e., goes out of scope). We might forget!!! Later you will see that you can tell the object to automatically invoke a very special method just before the object dies.

Exercise. Note that I have overwritten the default copy constructor:

```
class IntPointer
{
public:
    ...

    IntPointer(const IntPointer & intptr)
        : p_(new int)
    {
        *p_ = *(intptr.p_);
    }

    ...
};
```

Draw a memory model of what happens when you use this copy constructor, for instance, like this:

```
...
int main()
{
    IntPointer p(42);
    IntPointer q = p; // calls copy constructor
    return 0;
}
```

Check the definition of the default copy constructor. It copies values memberwise., i.e., the default copy constructor works like this:

```
class IntPointer
{
public:
    ...

    IntPointer(const IntPointer & intptr)
        : p_(intptr.p_)
    {}

    ...
};
```

What is the difference between this copy constructor (the default provided by the compiler if you do not have one) and the one that I wrote? (Hint: Draw a picture).

Now, what if I deallocate while using the default copy constructor?

```
class IntPointer
{
public:
    ...

    IntPointer(const IntPointer & intptr)
        : p_(intptr.p_)
    {}

    ...
};
```

```
...
int main()
{
    IntPointer p(42);
    IntPointer q = p;
    q.deallocate();
    p.deallocate();
    return 0;
}
```

Why?

Example and exercise: `IntArray`

Exercise. We frequently use a fixed size array with a length variable:

```
// What's missing here?

void print(int x[], int x_len)
{
    for (int i = 0; i < x_len; i++)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << std::endl;
}

int main()
{
    int x[1000];
    int x_len = 0;           // think of x as having
                             // nothing right now

    x[0] = 42;    x_len = 1; // think of x as having 1
                             // value
    print(x, x_len)

    x[1] = 1024; x_len = 2; // think of x as having 2
                             // value
    print(x, x_len)

    return 0;
}
```

Exercise. We should package up an int array with a length variable either using `struct` or `class`. Of course `class` has class! Here's the `IntArray` class:

```
class IntArray
{
public:
    IntArray();                 // Initialize length
                                // to 0.
    IntArray(const IntArray &); // Copy length and
                                // also the values!!
    int get_length();
    void set_length(int);
    void print();
private:
    int x_[1000];
    int length_;
};
```

and here's `main` to test the class:

```
// What's missing here?
```

```
int main()
{
    IntArray a;
    a.print();        // prints blank line;
                      // a.length_ is 0
    a.set_length(5);
    a.print();        // prints 5 random values;
                      // a.length_ is 5
    return 0;
}
```

Complete the class by implementing all the listed methods. Test your code by running it.

Exercise. There's still no way to set the values in the array. Add the following to the class:

```
class IntArray
{
public:

    ...

    int operator[](int i); // returns the value of x_[i]
};
```

Define the method (or rather, the operator) and then do this in `main()`:

```
int main()
{
    ...

    std::cout << a[0] << '\n'; // a[0] is a.operator[](0)

    return 0;
}
```

Exercise. Now do this:

```
int main()
{
    ...

    std::cout << a[0] << '\n'; // a[0] is a.operator[](0)
    a[0] = 42;

    return 0;
}
```

and ... you get a problem! It seems that getting `a[0]` works but modifying `a[0]` does not work. Fix it. (See `IntPointer`.)

Exercise. Once the `operator[]` is fixed you can do this:

```
int main()
{
    IntArray a;
```

```
    a.print(); // prints blank line; a.length_ is 0
    a.set_length(5);
    for (int i = 0; i < a.get_length(); i++)
    {
        a[i] = i * i;
    }
    a.print(); // prints 5 values; a.length_ is 5
    return 0;
}
```

Exercise. Add a sum method that returns the sum of the values of `a.x_[0], a.x_[1], ..., a.x_[a.length_ - 1]`. Test your code with this:

```
int main()
{
    IntArray a;
    a.print();   // prints blank line; a.length_ is 0
    a.set_length(5);
    for (int i = 0; i < a.get_length(); i++)
    {
        a[i] = i * i;
    }
    a.print();   // prints 5 values; a.length_ is 5
    std::cout << a.sum() << '\n';
    return 0;
}
```

Clearly you can also implement an array of doubles, of bools, of ...

Example and exercise: `IntDynArray`

The `IntArray` is great for packaging up all that you need to work with the concept of an array with changing length. Of course the actual maximum size of the array is fixed. (In the above case, it's fixed at 1000.) This is a waste if for instance you need only 10. And if you need to store more than 1000 values in an IntArray object, then you're out of luck.

Therefore it's better to **dynamically** request for what you need based on what happens when the program runs. Pointers to the rescue!!! Why? Because for array in the local scope the size must be constant:

```
int x[1000];                //OK
consts int SIZE = 20000;
int y[SIZE];                // OK
int n = 0;
std::cin >> n;
int z[n];                   // NO, NO, NO!!!
```

However arrays allocated in the free store can have variable sizes:

```
int * x = new int[1000];  //OK
// now use x[0], x[1], ..., x[999]
const int SIZE = 20000;
int * y = new int[SIZE];  // OK
// now use y[0], y[1], ..., y[19999]
int n = 0;
std::cin >> n;
int * z = new int[n];     // OK
// now use z[0], z[1], ..., z[n - 1]
delete [] z;
delete [] y;
delete [] x;
```

Pause … and study your notes on pointers again …

Exercise. Write the following dynamic integer array class. What I'm calling length in `IntArray`, I'm calling `size` here. The size of the array requested on the memory heap is `capacity`. Many of the methods requires minimum change from the version in `IntArray`.

```
class IntDynArray
{
public:
    IntDynArray(int capacity); // Allocate capacity
                               // number of integers
                               // for pointer x_. Set
                               // size_ to 0 and
                               // capacity_ to
                               // capacity.
    IntDynArray(const IntDynArray &);
    int get_size();
    void set_size(int);
    void get_capacity();
```

```
    void print();

private:
    int * x_;
    int size_;
    int capacity_;
};
```

Test your code with this:

```
int main()
{
    IntDynArray a(10); request 10 integers
    a.print(); // prints blank line; a.length is 0
    a.set_size(5);
    for (int i = 0; i < a.get_size(); i++)
    {
        a[i] = i * i;
    }
    a.print(); // prints 5 values
    std::cout << a.sum() << '\n';
    return 0;
}
```

Add a method to deallocate all memory allocated:

```
class IntDynArray
{
public:
    void deallocate();
};
```

and in `main` do this:

```
int main()
{
    ...
    a.deallocate();
    return 0;
}
```

Again, later you will see that you can define a method that will be called automatically whenever the object is about to die. All you need to do is to deallocate memory in this special method and we don't have to remember to deallocate anymore.

Clearly you can also implement a dynamical array of doubles, of chars, of …