Lecture 10: Potpourri, templates
Projects

• Hopefully you are done with 3E (3F?)
• Timeline for rest of term:
  — 3C: 11/15
  — 3D (optional): 11/15
  — 3E: 11/22 (a thinker, not much code)
  — 3F: 11/22 or 11/29
  — 3G: 11/29
  — 3H+4B: 12/8
  — Weds Dec 8 @ 10am: Hank does live code of project 3
Live code for final?
const
const

• const:
  – is a keyword in C and C++
  – qualifies variables
  – is a mechanism for preventing write access to variables
const example

```c
int main()
{
    const int X = 5;
}
```

The compiler enforces const ... just like public/private access controls
Efficiency

Are any of the three for loops faster than the others? Why or why not?

Answer: NumIterations is slowest … overhead for function calls.

Answer: X is probably faster than Y … compiler can do optimizations where it doesn’t have to do “i < X“ comparisons (loop unrolling)
const arguments to functions

- Functions can use const to guarantee to the calling function that they won’t modify the arguments passed in.

```c
struct Image {
    int width, height;
    unsigned char *buffer;
};

ReadImage(char *filename, Image &);
WriteImage(char *filename, const Image &);
```

read function can’t make the same guarantee

guarantees function won’t modify the Image
const pointers

• Assume a pointer named “P”

• Two distinct ideas:
  – P points to something that is constant
    • P may change, but you cannot modify what it points to via P
  – P must always point to the same thing, but the thing P points to may change.
const pointers

• Assume a pointer named “P”

• Two distinct ideas:
  – P points to something that is constant
    • P may change, but you cannot modify what it points to via P
  – P must always point to the same thing, but the thing P points to may change.
const pointers

int X = 4;
int *P = &X;

Idea #1:
violates const:

"*P = 3;"
OK:

"int Y = 5; P = &Y;"

pointer can change, but you can’t modify the thing it points to

Idea #2:
violates const:

"int Y = 5; P = &Y;"
OK:

"*P = 3;"

pointer can’t change, but you can modify the thing it points to
**Idea #3:**
violates const:

```
*P = 3;
```
```
int Y = 5; P = &Y;
```

OK:
none
**const pointers**

```c
int X = 4;
int *P = &X;
```

**Idea #1:**
violates const:
```
*P = 3;
```

**OK:**
```
int Y = 5;
P = &Y;
```

pointer can change, but you can’t modify the thing it points to

```
fawcett:330 childs$ cat const3.c
int main()
{
    int X = 5;
    int Y = 6;
    const int *P;
    P = &X;  // compiles
    P = &Y;  // compiles
    *P = 7;  // won’t compiles
}
fawcett:330 childs$ g++ const3.c
const3.C: In function ‘int main()’:
    const3.C:8: error: assignment of read-only location
```
const pointers

int X = 4;
int *P = &X;

Idea #2:
violates const:
“int Y = 5; P = &Y;”
OK:
“*P = 3;”

pointer can’t change, but you can modify the thing it points to
const pointers

Idea #3: violates const:

```
"*P = 3;"
"int Y = 5; P = &Y;"
```

OK:

none

pointer can’t change, and you can’t modify the thing it points to
const usage

• class Image;
• const Image *ptr;
  – Used a lot: offering the guarantee that the function won’t change the Image ptr points to
• Image * const ptr;
  – Helps with efficiency. Rarely need to worry about this.
• const Image * const ptr;
  – Interview question!!
Very common issue with const and objects

How does compiler know GetNumberOfPixels doesn’t modify an Image?

We know, because we can see the implementation.

But, in large projects, compiler can’t see implementation for everything.
const functions with objects

If a class method is declared as const, then you can call those methods with pointers.
mutable

• mutable: special keyword for modifying data members of a class
  – If a data member is mutable, then it can be modified in a const method of the class.
  – Comes up rarely in practice.
globals
globals

• You can create global variables that exist outside functions.

```c
#include <stdio.h>
int X = 5;

int main()
{
    printf("X is %d\n", X);
}
```

```
fawcett:Documents childs$ g++ global1.C
fawcett:Documents childs$ ./a.out
X is 5
fawcett:Documents childs$
```
• global variables are initialized before you enter main
Storage of global variables...

- Global variables are stored in a special part of memory — “data segment” (not heap, not stack)
- If you re-use global names, you can have collisions

```c
#include <stdio.h>

int double(int Y) {
    return 2*Y;
}
```

```
fawcett:Documents child$ $ cat file1.C
int X = 6;

int main() {
}

fawcett:Documents child$ $ g++ -c file1.C

fawcett:Documents child$ $ cat file2.C
int X = 7;

int doubler(int Y)
{
    return 2*Y;
}

fawcett:Documents child$ $ g++ -c file2.C

fawcett:Documents child$ $ g++ file1.o file2.o
ld: duplicate symbol _X in file2.o and file1.o
collect2: ld returned 1 exit status
```
Externs: mechanism for unifying global variables across multiple files

```
extern int count;

int doubler(int Y)
{
    count++;
    return 2*Y;
}
```

```
fawcett:330 childs$ cat file2.C
extern int count;
```
static

• static memory: third kind of memory allocation
  – reserved at compile time
• contrasts with dynamic (heap) and automatic (stack) memory allocations
• accomplished via keyword that modifies variables

There are three distinct usages of statics
static usage #1: persistency within a function

```c
#include <stdio.h>

int fibonacci() {
    static int last2 = 0;
    static int last1 = 1;
    int rv = last1+last2;
    last2 = last1;
    last1 = rv;
    return rv;
}

int main() {
    int i;
    for (int i = 0; i < 10; i++)
        printf("%d\n", fibonacci());
}
```

static usage #2: making global variables be local to a file

I have no idea why the static keyword is used in this way.
static usage #3: making a singleton for a class

fawcett:Downloads childs$ cat static3.C
#include <iostream>

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

class MyClass
{
  public:
    MyClass() { numInstances++; }
    virtual ~MyClass() { numInstances--; }

    int GetNumInstances(void) { return numInstances; }

  private:
    int numInstances;
};

int main()
{
  MyClass *p = new MyClass[10];
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
  delete [] p;
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
fawcett:Downloads childs$ g++ static3.C
fawcett:Downloads childs$ ./a.out
Num instances = 1
Num instances = 0
static usage #3: making a singleton for a class

```cpp
#include <iostream>

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

class MyClass
{
  public:
    MyClass() 
    virtual ~MyClass() 
    int GetNumInstances() 
  private:
    static int numInstances; 
};

int main()
{
  delete [] p;
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
```

We have to tell the compiler where to store this static.

What do we get?

```
fawcett:Downloads child3$ g++ static3.C
Undefined symbols:
  "MyClass::numInstances", referenced from:
    MyClass::MyClass() in ccoao8Hf.o
    MyClass::MyClass() in ccoao8Hf.o
    MyClass::GetNumInstances() in ccoao8Hf.o
  MyClass::~MyClass() in ccoao8Hf.o
  MyClass::~MyClass() in ccoao8Hf.o
  MyClass::~MyClass() in ccoao8Hf.o
  MyClass::~MyClass() in ccoao8Hf.o
  MyClass::~MyClass() in ccoao8Hf.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```
static usage #3: making a singleton for a class

```cpp
#include <iostream>

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

class MyClass
{
  public:
    MyClass() { numInstances++; }
    virtual ~MyClass() { numInstances--; }

    int GetNumInstances(void) { return numInstances; }

  private:
    static int numInstances;

};

int MyClass::numInstances = 0;

int main()
{
  MyClass *p = new MyClass[10];
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
  delete [] p;
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
Static data members and static methods are useful and they are definitely used in practice.
Scope
• I saw this bug quite a few times...

The compiler will sometimes have multiple choices as to which variable you mean.

It has rules to make a decision about which one to use.

This topic is referred to as "scope".

class MyClass
{
    public:
    
    void SetValue(int);

    private:
    
    int X;
};

void MyClass::SetValue(int X)
{
    X = X;
}
```cpp
int X = 0;

class MyClass
{
    public:
        MyClass() { X = 1; };

        void SetValue(int);

    private:
        int   X;
};

void MyClass::SetValue(int X)
{
    int X = 3;
    cout << "X is " << X << endl;
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
```

This one won’t compile.

The compiler notices that you have a variable called X that “shadows” the argument called X.
This one will compile ... the compiler thinks that you made a new scope on purpose.

So what does it print?

Answer: 3
What does this one print?  

Answer: 2
int X = 0;

class MyClass
{
    public:
        MyClass() { X = 1; };

    void SetValue(int);

private:
    int    X;
};

void MyClass::SetValue(int X)
{
    int X = 3;
    cout << "X is " << X << endl;
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
int X = 0;
class MyClass {
    public:
        MyClass() { X = 1; }
        void SetValue(int);
    private:
        int X;
};
void MyClass::SetValue(int X) {
    int X = 3;
    cout << "X is " << X << endl;
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
Scope Rules

- The compiler looks for variables:
  - inside a function or block
  - function arguments
  - data members (methods only)
  - globals
Shadowing

- Shadowing is a term used to describe a “subtle” scope issue.
  - ... i.e., you have created a situation where it is confusing which variable you are referring to

```cpp
class Sink{
    public:
        void SetInput(Image *i) { input = i; };
    protected:
        Image *input;
};

class Writer : public Sink{
    public:
        void Write(void) { /* write input */ };
    protected:
        Image *input;
};

int main()
{
    Writer writer;
    writer.SetInput(image);
    writer.Write();
}
```
Overloading Operators

• NOTE: I lectured on this some, but it was informal. These slides formally capture the ideas we discussed.
C++ lets you define operators

• You declare a method that uses an operator in conjunction with a class
  – +, -, /, !, ++, etc.

• You can then use operator in your code, since the compiler now understands how to use the operator with your class

• This is called “operator overloading”
  – ... we are overloading the use of the operator for more than just the simple types.
Example of operator overloading

class MyInt
{
    public:
        MyInt(int x) { myInt = x; }
        MyInt& operator++();
    int     GetValue(void) { return myInt; }

    protected:
        int     myInt;
};

MyInt &
MyInt::operator++()
{
    myInt++; return *this;
}

int main()
{
    MyInt mi(6);
    ++mi;
    ++mi;
    printf("Value is %d\n", mi.GetValue());
}

fawcett:330 childs$ ./a.out
Value is 8

Declare operator ++ will be overloaded for MyInt

Call operator ++ on MyInt.

Define operator ++ for MyInt
More operator overloading

```cpp
fawcett:330 childs$ cat oostream.C
#include <iostream>

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

class Image
{
  public:
    Image();

    friend ostream& operator<<(ostream &os, const Image &);

  private:
    int width, height;
    unsigned char *buffer;
};

Image::Image()
{
  width = 100;
  height = 100;
  buffer = NULL;
}

ostream &
operator<<(ostream &out, const Image &img)
{
  out << img.width << "x" << img.height << endl;
  if (img.buffer == NULL)
    out << "No buffer allocated!" << endl;
  else
    out << "Buffer is allocated!" << endl;
}

int main()
{
  Image img;
  cout << img;
}

fawcett:330 childs$ g++ oostream.C
fawcett:330 childs$ ./a.out
100x100
No buffer allocated!
```
Beauty of inheritance

- ostream provides an abstraction
  - That’s all Image needs to know
    - it is a stream that is an output
    - You code to that interface
    - All ostream’s work with it

```c++
int main()
{
    Image img;
    cerr << img;
}
```

```
fawcett:330 child$ ./a.out
100x100
No buffer allocated!
```

```
int main()
{
    ofstream ofile("output_file");
    ofile << img;
}
```

```
fawcett:330 child$ g++ oostream.C
fawcett:330 child$ ./a.out
fawcett:330 child$ cat output_file
100x100
No buffer allocated!
```
class Image
{
    public:
        Image();
        void SetSize(int w, int h);
    friend ostream& operator<<(ostream &os, const Image &);
    Image & operator=(const Image &);

    private:
        int width, height;
        unsigned char *buffer;
};

void Image::SetSize(int w, int h)
{
    if (buffer != NULL)
        delete [] buffer;
    width = w;
    height = h;
    buffer = new unsigned char[3*width*height];
}

int main()
{
    Image img1, img2;
    img1.SetSize(200, 200);
    cout << "Image 1: " << img1;
    cout << "Image 2: " << img2;
    img2 = img1;
    cout << "Image 1: " << img1;
    cout << "Image 2: " << img2;
}
let’s do this again...

```cpp
ostream &
operator<<(ostream &out, const Image &img)
{
    out << img.width << "x" << img.height << endl;
    if (img.buffer == NULL)
        out << "No buffer allocated!" << endl;
    else
        out << "Buffer is allocated, and value is "
            << (void *) img.buffer << endl;

    return out;
}
```

```
fawcett:330 childs$ ./a.out
Image 1:200x200
Buffer is allocated, and value is 0x100800000
Image 2:0x0
No buffer allocated!
Image 1:200x200
Buffer is allocated, and value is 0x100800000
Image 2:200x200
Buffer is allocated, and value is 0x10081e600
```
let’s do this again...

class Image
{
    public:
        Image();
        void SetSize(int w, int h);
    friend ostream& operator<<(ostream &os, const Image &);
    // Image & operator=(const Image &);
    private:
        int width, height;
        unsigned char *buffer;
};

int main()
{
    Image img1, img2;
    img1.SetSize(200, 200);
    cout << "Image 1:" << img1;
    cout << "Image 2:" << img2;
    img2 = img1;
    cout << "Image 1:" << img1;
    cout << "Image 2:" << img2;
}

fawcett:330 childs$ g++ assignment_op.C
fawcett:330 childs$
C++ defines a default assignment operator for you

• This assignment operator does a bitwise copy from one object to the other.
• Does anyone see a problem with this?

```
fawcett:330 childs$ ./a.out
Image 1:200x200
Buffer is allocated, and value is 0x1008000000
Image 2:0x0
No buffer allocated!
Image 1:200x200
Buffer is allocated, and value is 0x1008000000
Image 2:200x200
Buffer is allocated, and value is 0x1008000000
```

This behavior is sometimes OK and sometimes disastrous.
Copy constructors: same deal

• C++ automatically defines a copy constructor that does bitwise copying.

• Solutions for copy constructor and assignment operators:
  – Re-define them yourself to do “the right thing”
  – Re-define them yourself to throw exceptions
  – Make them private so they can’t be called
Upcasting and Downcasting

• **Upcast:** treat an object as the base type
  – We do this all the time!
  – Treat a Rectangle as a Shape

• **Downcast:** treat a base type as its derived type
  – We don’t do this one often
  – Treat a Shape as a Rectangle
    • You better know that Shape really is a Rectangle!!
Upcasting and Downcastcating

```cpp
class A {
};

class B : public A {
    public:
        B() { myInt = 5; }
        void Printer(void) { cout << myInt << endl; }

    private:
        int myInt;
};

void Downcaster(A *a) {
    B *b = (B *) a;
    b->Printer();
}

int main() {
    A a;
    B b;

    Downcaster(&b); // no problem
    Downcaster(&a); // no good
}
```

```
fawcett:330 childs$ g++ downcaster.C
fawcett:330 childs$ ./a.out
5
-1074118656
```
Upcasting and Downcasting

• C++ has a built-in facility to assist with downcasting: dynamic_cast
• I personally haven’t used it a lot, but it is used in practice
• Ties in to std::exception
Default Arguments

```cpp
void Foo(int X, int Y = 2)
{
    cout << "X = " << X << ", Y = " << Y << endl;
}

int main()
{
    Foo(5);
    Foo(5, 4);
}
```

default arguments: compiler pushes values on the stack for you if you choose not to enter them
Booleans

- New simple data type: bool (Boolean)
- New keywords: true and false

```c
int main()
{
    bool b = true;
    cout << "Size of boolean is " << sizeof(bool) << endl;
}
```

fawcett:330 childs$ g++ Boolean.C
fawcett:330 childs$ ./a.out
Templates
Motivation

```
int Doubler(int X) { return 2*X; };
float Doubler(float X) { return 2*X; };

int main()
{
    int X = 2;
    float Y = 2.6;
    cout << "2*X = " << Doubler(X) << ", 2*Y = " << Doubler(Y) << endl;
}
fawcett:330 childs$ g++ logger_defines.C
fawcett:330 childs$ ./a.out
2*X = 4, 2*Y = 5.2

fawcett:330 childs$ nm a.out
0000000100000d7a s stub helpers
00000001000010b0 D _NXArgc
00000001000010b8 D _NXArgv
0000000100000ac7 t __GLOBAL__I__Z7Doubleri
0000000100000a84 t __Z41__static_initialization_and_destruction_0ii
0000000100000b26 T __Z7Doublerf
0000000100000b18 T __Z7Doubleleri
```
Motivation
```cpp
#include <iostream>

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

template <class T> T Doubler(T X) { return 2*X; }

int main()
{
    int   X = 2;
    float Y = 2.6;
    cout << "2*X = " << Doubler(X) << ", 2*Y = " << Doubler(Y) << endl;
}
```
Will now do an example to compare templates and virtual functions

• Will take some buildup...
class Money
{
  public:
    Money(int d, int c) { dollars = d; cents = c; }
    bool operator<(const Money &m);

  private:
    int dollars;
    int cents;
};

bool Money::operator<(const Money &m)
{
  if (dollars < m.dollars) return true;
  if (dollars == m.dollars) return (cents < m.cents);
  return false;
}

int main()
{
  Money m(6, 85);
  Money m2(6, 25);
  bool lt = m < m2;
  cerr << "LT = " << lt << endl;
  lt = m2 < m;
  cerr << "LT = " << lt << endl;
}

C02LN00GFD58:330 hank$ g++ money.C
C02LN00GFD58:330 hank$ ./a.out
LT = 0
LT = 1
class LicensePlate
{
public:
    LicensePlate(char c1, char c2, char c3,
                 int i1, int i2, int i3)
    {
        letters[0] = c1;
        letters[1] = c2;
        letters[2] = c3;
        numbers[0] = i1;
        numbers[1] = i2;
        numbers[2] = i3;
    }

    bool operator<(const LicensePlate &rhs)
    {
        for (int i = 0; i < 3; i++)
        {
            if (letters[i] < rhs.letters[i])
                return true;
            if (letters[i] > rhs.letters[i])
                return false;
        }
        for (int i = 0; i < 3; i++)
        {
            if (numbers[i] < rhs.numbers[i])
                return true;
            if (numbers[i] > rhs.numbers[i])
                return false;
        }
        // equal
        return false;
    }
private:
    char    letters[3];
    int     numbers[3];
};

int main()
{
    LicensePlate lp1('a', 'b', 'c', 4, 5, 6);
    LicensePlate lp2('c', 'b', 'a', 6, 5, 4);
    bool lt = lp1 < lp2;
    cerr << "LT = " << lt << endl;
    lt = lp2 < lp1;
    cerr << "LT = " << lt << endl;
}
Sorting With Templates

template <class T>
void Sort(T **X, int nX)
{
    for (int i = 0; i < nX; i++)
        for (int j = i+1; j < nX; j++)
            if (*X[j] < *X[i])
                { 
                    T *tmp = X[j];
                    X[j] = X[i];
                    X[i] = tmp;
                }
}

int main()
{
    Money m1(6, 85);
    Money m2(6, 25);
    Money m3(4, 25);
    Money m4(5, 25);
    LicensePlate lp1('a', 'b', 'c', 4, 5, 6);
    LicensePlate lp2('c', 'b', 'a', 6, 5, 4);
    LicensePlate lp3('c', 'd', 'a', 6, 5, 4);
    LicensePlate lp4('b', 'b', 'a', 6, 5, 4);
    Money *money_list[4] = { &m1, &m2, &m3, &m4 };
    LicensePlate *lp_list[4] = { &lp1, &lp2, &lp3, &lp4 };
    Sort(money_list, 4);
    Sort(lp_list, 4);
    for (int i = 0; i < 4; i++)
        cout << i << ":: $" << money_list[i]->dollars << "." << money_list[i]->cents << endl;
    for (int i = 0; i < 4; i++)
        { 
            cout << i << ": ";
            PrintLicensePlate(lp_list[i]);
            cout << endl;
        }
Doing the same with inheritance

class Sortable
{
    public:
        virtual bool operator<(const Sortable *) = 0;
};

class LicensePlate : public Sortable
{
    public:
        LicensePlate(char c1, char c2, char c3,
                      int i1, int i2, int i3)
        {
            letters[0] = c1;
            letters[1] = c2;
            letters[2] = c3;
            numbers[0] = i1;
            numbers[1] = i2;
            numbers[2] = i3;
        }

        bool operator<(const Sortable *);

    public:
        char    letters[3];
        int     numbers[3];
};

void Sort(Sortable **X, int nX)
{
    for (int i = 0 ; i < nX ; i++)
        for (int j = i+1 ; j < nX ; j++)
            if (*X[j] < X[i])
                {
                    Sortable *tmp = X[j];
                    X[j] = X[i];
                    X[i] = tmp;
                }
}

int main()
{
    LicensePlate lp1('a', 'b', 'c', 4, 5, 6);
    LicensePlate lp2('c', 'b', 'a', 6, 5, 4);
    LicensePlate lp3('c', 'd', 'a', 6, 5, 4);
    LicensePlate lp4('b', 'b', 'a', 6, 5, 4);

    Sortable *lp_list[4] = { &lp1, &lp2, &lp3, &lp4 };
    Sort(lp_list, 4);
    for (int i = 0 ; i < 4 ; i++)
        {
            cout << i << " : ";
            PrintLicensePlate((LicensePlate *)lp_list[i]);
            cout << endl;
        }
}
Templates vs Virtual Functions

• Virtual functions:
  – Had to affect inheritance hierarchy
  – Overhead in function call (virtual function table)

• Templates:
  – Did not need to affect inheritance hierarchy, although function names had to coincide
  – No additional overhead (resolved at compile time)
Standard Template Library
Standard Template Library

- Standard Template Library: STL
- Many, many templated types
- Can ease your programming burden
- Can also hide significant performance issues
  - And you use C/C++ for performance
- My recommendation: use with caution for code that needs to be performant
#include <vector>

using std::vector;

int main()
{
    vector<int> intArray(2);
    intArray[0] = 0;
    intArray[1] = 1;
    intArray.push_back(1);
    intArray.push_back(2);
    intArray.push_back(3);
    intArray.push_back(5);
    cout << "Size is " << intArray.size() << endl;
    cout << "Last val of Fib is " << intArray[5] << endl;
}

C02LN00GFD58:330 hank$ g++ vector.C
C02LN00GFD58:330 hank$ ./a.out
Size is 6
Last val of Fib is 5
std::vector

• Always has the amount of memory you need
• Many STL algorithms work on it
• Memory allocation:
  – If you don’t have enough, double it
    • (can be a big overestimation)
• Overhead in access
  – Maybe not a big deal if data-intensive?
#include <map>
#include <string>

using std::map;
using std::string;

int main()
{
    map<string, int> ageLookup;
    ageLookup["Hank"] = 37;
    ageLookup["Charlotte"] = 11;
    ageLookup["William"] = 9;

    cout << "Hank's age is " << ageLookup["Hank"] << endl;
    cout << "Carissa's age is " << ageLookup["Carissa"] << endl;
}

C02LN00GFD58:330 hank$ g++ map.c
C02LN00GFD58:330 hank$ ./a.out
Hank's age is 37
Carissa's age is 0
C++ Strings
(not a template thing): String

- C++ string class is very useful
- Great implementation of a class that encapsulates a string

```cpp
#include <string>

using std::string;

int main()
{
    string str = "Hello";
    str += " world";
    cout << str << endl;
}
```

```bash
C02LN00GFD58:330 hank$ g++ string.C
C02LN00GFD58:330 hank$ ./a.out
Hello world
```
## String methods

### Iterators:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin</code></td>
<td>Return iterator to beginning</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>end</code></td>
<td>Return iterator to end</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>rbegin</code></td>
<td>Return reverse iterator to reverse beginning</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>Return reverse iterator to reverse end</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>cbegin</code></td>
<td>Return const_iterator to beginning</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>cend</code></td>
<td>Return const_iterator to end</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>crbegin</code></td>
<td>Return const_reverse_iterator to reverse beginning</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>crend</code></td>
<td>Return const_reverse_iterator to reverse end</td>
<td>(public member function)</td>
</tr>
</tbody>
</table>

### Capacity:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size</code></td>
<td>Return length of string</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Return length of string</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>Return maximum size of string</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>resize</code></td>
<td>Resize string</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>capacity</code></td>
<td>Return size of allocated storage</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>reserve</code></td>
<td>Request a change in capacity</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>clear</code></td>
<td>Clear string</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>empty</code></td>
<td>Test if string is empty</td>
<td>(public member function)</td>
</tr>
<tr>
<td><code>shrink_to_fit</code></td>
<td>Shrink to fit</td>
<td>(public member function)</td>
</tr>
</tbody>
</table>
# String methods

## Element access:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator[]</code></td>
<td>Get character of string (public member function)</td>
</tr>
<tr>
<td>at</td>
<td>Get character in string (public member function)</td>
</tr>
<tr>
<td>back</td>
<td>Access last character (public member function)</td>
</tr>
<tr>
<td>front</td>
<td>Access first character (public member function)</td>
</tr>
</tbody>
</table>

## Modifiers:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator+=</code></td>
<td>Append to string (public member function)</td>
</tr>
<tr>
<td>append</td>
<td>Append to string (public member function)</td>
</tr>
<tr>
<td>push_back</td>
<td>Append character to string (public member function)</td>
</tr>
<tr>
<td>assign</td>
<td>Assign content to string (public member function)</td>
</tr>
<tr>
<td>insert</td>
<td>Insert into string (public member function)</td>
</tr>
<tr>
<td>erase</td>
<td>Erase characters from string (public member function)</td>
</tr>
<tr>
<td>replace</td>
<td>Replace portion of string (public member function)</td>
</tr>
<tr>
<td>swap</td>
<td>Swap string values (public member function)</td>
</tr>
<tr>
<td>pop_back</td>
<td>Delete last character (public member function)</td>
</tr>
</tbody>
</table>

## String operations:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_str</td>
<td>Get C string equivalent (public member function)</td>
</tr>
<tr>
<td>data</td>
<td>Get string data (public member function)</td>
</tr>
<tr>
<td>get_allocator</td>
<td>Get allocator (public member function)</td>
</tr>
<tr>
<td>copy</td>
<td>Copy sequence of characters from string (public member function)</td>
</tr>
<tr>
<td>find</td>
<td>Find content in string (public member function)</td>
</tr>
<tr>
<td>rfind</td>
<td>Find last occurrence of content in string (public member function)</td>
</tr>
<tr>
<td>find_first_of</td>
<td>Find character in string (public member function)</td>
</tr>
<tr>
<td>find_last_of</td>
<td>Find character in string from the end (public member function)</td>
</tr>
<tr>
<td>find_first_not_of</td>
<td>Find absence of character in string (public member function)</td>
</tr>
<tr>
<td>find_last_not_of</td>
<td>Find non-matching character in string from the end (public member function)</td>
</tr>
<tr>
<td>substr</td>
<td>Generate substring (public member function)</td>
</tr>
<tr>
<td>compare</td>
<td>Compare strings (public member function)</td>
</tr>
</tbody>
</table>