Introduction to C++

Lecture 7

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

Comparison between C and C++ (1)

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

Bjarne Stroustrup – C++ creator

Comparison between C and C++ (2)

In C, you merely shoot yourself in the foot.

In C++, you accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical care is impossible, because you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me, over there."

A joke from Usenet

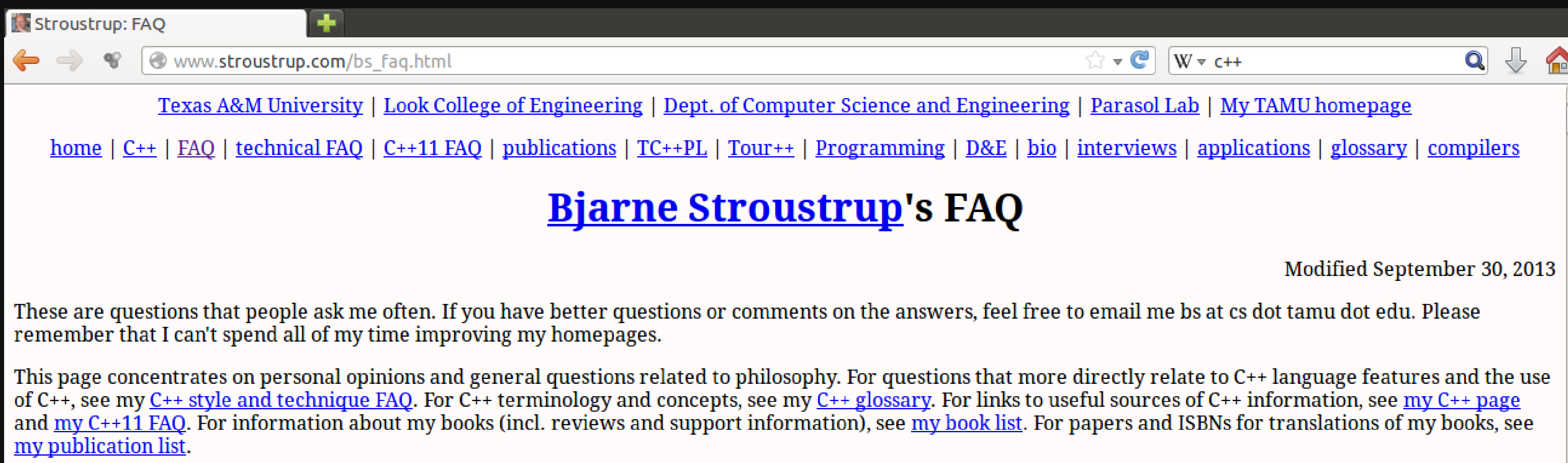
Where it came from?

C++ development started early in 1979 at AT&T by **Bjarne Stroustrup**.

Where it came from?

Bjarne Stroustrup's FAQ

http://www.stroustrup.com/bs_faq.html



Some Features of C++

- **Multiparadigm**.
- Meant to be **cross-plataform**.
- **Weakly typed**.
- Allows for **generic programming**:
  - Parameterization of types and algorithms.
- Has **polymorphism**.
- The standard requires the **Standard Library**.
- Etc.

C++ Compilers

- There are **several C++ compilers** around. Some examples are:
  - Open Watcom C/C++.
  - CodeWarrior.
  - GCC.
- More C++ compilers on:
  - https://en.wikipedia.org/wiki/List_of_compilers#C.2B.2B_compilers

Versions of C++

- Since its creation, C++ has undergone several **improvements**.
- Resulting **versions** have been **published** as **standards**:
  - C++98/C++03
    - ISO/IEC 14882:1998 (amended by ISO/IEC 14882:2003).
    - **`g++`** option: **`-ansi`** or **`-std=c++98`** or **`-std=c++03`**.
  - C++11
    - ISO/IEC 14882:2011.
    - **`g++`** option: **`-std=c++11`**
  - C++14
    - ISO/IEC 14882:2014.
    - **`g++`** option: **`-std=c++14`**

During this course we will use the C++11 (**ISO/IEC 14882:2011**) dialect of C++.
`g++` can be instructed to follow the above standard with the option **`-std=c++11`**.

Very Simple C vs. C++ Example

The two programs below just print out the famous "**Hello World!**" message.

C version

example_c.c

```c
#include "stdio.h"

int main( void )
{
    printf( "Hello world!\n" );

    return 0;
}
```

```
~$ gcc -std=c99 ... main.c -o main
```

C++ version

example_cpp.cpp

```cpp
#include <iostream>

int main( void )
{
  std::cout << "Hello world!\n";

  return 0;
}
```

```
~$ g++ -std=c++11 ... main.cpp -o main
```

```
-Wall -Werror -pedantic
```

C++ Data Types

- Basic types
  - **`char`**, **`int`**, **`float`**, **`bool`**, **`double`**, pointers.
- Structured types
  - Arrays, structs, unions.
- User defined types
  - Created with **`typedef`**, **`using`**.
  - Classes.

Some Basic New Concepts of C++

- **Classes** and **objects**.
- Member **functions**.
- **Private**, **public** and **protected** members.
- Among others...

Let's Start with a Problem

Write a program that implements **vectors** of four elements (floats) and the following operations between them: **copy** and **sum**.

C-based Solution

We might start by defining the **data structure** in charge of storing the 4-float vector:

vector.c

```
typedef float Vector[4];
...

int main( void )
{
    Vector a, b, c;
    ...
```

Type definition.

Variable declaration.

C-based Solution

Once we have defined the vector data structure, we can implement the **copy** and the **sum** operations:

vector.c — `CopyVec()` **definition**

```
...
void CopyVec( float* v1, float* v2 )
{
    v2[0] = v1[0];
    v2[1] = v1[1];
    v2[2] = v1[2];
    v2[3] = v1[3];
}
...
int main( void )
{
    Vector a, b;
    ...
    CopyVec( a, b );
    ...
```

vector.c — `SumVec()` **definition**

```
...
void SumVec( float* v1, float* v2, float* v3 )
{
    v3[0] = v1[0] + v2[0];
    v3[1] = v1[1] + v2[1];
    v3[2] = v1[2] + v2[2];
    v3[3] = v1[3] + v2[3];
}
...
int main( void )
{
    Vector a, b, c;
    ...
    SumVec( a, b, c );
    ...
```

C-based Solution

This is the complete C-based solution for the proposed problem

vector.c

```
typedef float Vector[4];

void CopyVec( float* v1, float* v2 ) {
    v1[0] = v2[0]; v1[1] = v2[1];
    v1[2] = v2[2]; v1[3] = v2[3];
}

void SumVec( float* v1, float* v2, float* v3 ) {
    v1[0] = v2[0] + v3[0]; v1[1] = v2[1] + v3[1];
    v1[2] = v2[2] + v3[2]; v1[3] = v2[3] + v3[3];
}

int main( void ){
    Vector a, b, c;
    // initialize vectors 'a' and 'b' here....
    CopyVec( a, b );
    SumVec( a, b, c );

    return 0;
}
```

New Requirement

Now we want to keep, for **each** vector, the **average** of its values. If the vector is modified, the average must be **updated accordingly** in order to maintain the **consistency**.

C-based Solution

Now, we've decided that a **`struct`** will be responsible for storing all vector-related data (vector values and the average):

vector.c

```
struct Vector{
    float v[4];
    float mean;
};
...

int main( void )
{
    struct Vector a, b, c;
    ...
```

Type definition.

C-based Solution

The **`mean`** field must be updated each time the vector contents are changed. This will be accomplished by the new function **`MeanVec()`**.

vector.c

```
...

void MeanVec( struct Vector* v ) {
    v->mean = ( v->v[0] + v->v[1] + v->v[2] + v->v[3] ) * 0.25f;
}

...

int main( void )
{
    struct Vector a;
    ... // initialization of the variable a
    MeanVec( &a );
    ...
```

C-based Solution

- `SumVec()` and `CopyVec()` must be **rewritten** in order to accommodate the requirements of the **new vector structure**.
- Additionally, these functions **must update** the newly created `mean` field consistently.

C-based Solution

Rewritting **CopyVec()**:

vector.c (old)

```
...
void CopyVec( float* v1, float* v2 ) {
    v1[0] = v2[0];
    v1[1] = v2[1];
    v1[2] = v2[2];
    v1[3] = v2[3];
}
...
```

Pointers to the structs.

vector.c (new)

```
...

void CopyVec( struct Vector* v1, struct Vector* v2 ) {
    v1->v[0] = v2->v[0];
    v1->v[1] = v2->v[1];
    v1->v[2] = v2->v[2];
    v1->v[3] = v2->v[3];
    v1->mean = v2->mean;
}
...
```

Makes a copy of mean.

C-based Solution

Rewritting **SumVec()**:

vector.c (old)

```
...
void SumVec( float* v1, float* v2, float* v3 ) {
    v3[0] = v1[0] + v2[0];
    v3[1] = v1[1] + v2[1];
    v3[2] = v1[2] + v2[2];
    v3[3] = v1[3] + v2[3];
}
...
```

Pointers to the structs.

vector.c (new)

```
...

void SumVec( struct Vector* v1, struct Vector* v2, struct Vector* v3 ) {
    v1->v[0] = v2->v[0] + v3->v[0];
    v1->v[1] = v2->v[1] + v3->v[1];
    v1->v[2] = v2->v[2] + v3->v[2];
    v1->v[3] = v2->v[3] + v3->v[3];
    v1->mean = v2->mean + v3->mean;
}
...
```

Recompute mean.

C++-based Solution

First, we create a class containing all the vector related data:

vector.c

```
struct Vector {
     float v[4];
     float mean;
};
...

int main( void )
{
     struct Vector a, b, c;
     ...
```

vector.cpp

```
class Vector {
public:

     float v_[4];
     float mean_;
};
...
int main( void )
{
     Vector a, b, c;
     ...
```

Class definiton.

Objects: instances of the class.

Class: type.

C++-based Solution

Adding a **constructor** and a **destructor** to the Vector class:

vector.cpp

```
class Vector {
public:
    Vector( void )
    {
        v_[0]=0.0f; v_[1]=0.0f;
        v_[2]=0.0f; v_[3]=0.0f;
        mean_ = 0.0f;
    }

    ~Vector( void ){}

    float v_[4];
    float mean_;
};
...
```

Constructor

Destructor

C++-based Solution

Adding **function members** to the Vector class:

vector.cpp

```
class Vector {
public:
    ...
    void setMean( void ) {
        mean_ = (v_[0] + v_[1] + v_[2] + v_[3]) * 0.25f;
    }

    void copyFrom( Vector a ) {
        v_[0] = a.v_[0]; v_[1] = a.v_[1]; v_[2] = a.v_[2]; v_[3] = a.v_[3];
        mean_ = a.mean_;
    }

    void sum( Vector a )  {
        v_[0] += a.v_[0]; v_[1] += a.v_[1]; v_[2] += a.v_[2]; v_[3] += a.v_[3];
        mean_ += a.mean_;
    }

    float v_[4];
    float mean_;
};
...
```

Simulating C++ Classes With C

C++ class example.

class_cpp.cpp

```
class CPPClass {
public:
  int x_;
  int y_;

  void set( int a, int b ) {
    this->x_ = a;
    this->y_ = b;
  }

  int getX( void ) {
    return this->x_;
  }
};

CPPClass cppobj;

int main( void )
{
  cppobj.set( 10, 20 );
  return cppobj.getX();
}
```

What about the **assembly** code **generated** for each program?

It seems that **g++** actually implements member functions as ordinary C functions with an extra (implicit) parameter: a pointer to a structure that contains the class data.

C code that simulates `class_cpp.cpp`.

class_c.c

```
struct CClass {
  int x_;
  int y_;
};

void Set( struct CClass *this,
          int a, int b ) {
  this->x_ = a;
  this->y_ = b;
}

int GetX( struct CClass *this ) {
  return this->x_;
}

struct CClass cobj;

int main( void ) {
  Set( &cobj, 10, 20 );
  return GetX( &cobj );
}
```

```
~$ gcc -std=c99 -O0 -E class_c.c > class_c.i
~$ gcc -std=c99 -O0 -fno-asynchronous-unwind-tables -S class_c.i
```

```
~$ g++ -std=c++11 -O0 -E class_cpp.cpp > class_cpp.i
~$ g++ -std=c++11 -O0 -fno-asynchronous-unwind-tables -fno-dwarf2-cfi-asm -S class_cpp.i
```

The Current C++ Solution

vector.cpp

```
class Vector {
public:
  Vector( void ) {
    v_[0]=0.0f; v_[1]=0.0f; v_[2]=0.0f; v_[3]=0.0f;
    mean_ = 0.0f;
  };

  ~Vector( void ){};

  void setMean( void ) {
    mean_ = (v_[0] + v_[1] + v_[2] + v_[3]) * 0.25f;
  }

  void copyFrom( Vector a ) {
    v_[0] = a.v_[0]; v_[1] = a.v_[1]; v_[2] = a.v_[2]; v_[3] = a.v_[3];
    mean_ = a.mean_;
  }

  void sum( Vector a)  {
    v_[0] += a.v_[0]; v_[1] += a.v_[1]; v_[2] += a.v_[2]; v_[3] += a.v_[3];
    mean_ += a.mean_;
  }

  float v_[4];
  float mean_;
};
...
```

vector.cpp

```
...
int main(void)
{
  Vector a, b, c;

  a.v_[0] = 0.0f;
  a.v_[1] = 1.0f;
  a.v_[2] = 2.0f;
  a.v_[3] = 3.0f;
  a.setMean();

  b.copyFrom( a );
  c.copyFrom( b );
  c.sum( b );

  return 0;
}
```

Data members can be **freely accessed** from **outside the class** through the "." operator (this might lead to **inconsistencies**)! **How to avoid that?**

Access Specifiers

An access-specifier specifies the access rules for members following it until the end of the class or until another access-specifier is encountered.

C++11 Spec

There are three access specifiers:

- **`public`.**
- **`private`.**
- **`protected` (will be discussed later).**

vector.cpp

```
class Vector {
public:
  Vector( void ) { ... };

  ~Vector( void ){};

  void setMean( void ) { ... }

  void copyFrom( Vector a ) { ... }

  void sum( Vector a )  { ... }

private:
  float v_[4];
  float mean_;
};
...
```

Now **`v_`** and **`mean_`** are **`private`**, i.e, they can **only** be **accessed** by **member functions** of the class!

However, **how** can we now **access** the **contents** of the **`v_`** and **`mean_`** data members?

Setters / Mutators, Getters / Accessors

vector.cpp

```
class Vector {
public:
  ...
  void setValues( float v0, float v1, float v2, float v3 ) {
    v_[0] = v0;
    v_[1] = v1;
    v_[2] = v2;
    v_[3] = v3;
    setMean();
  }

  void getValues( float *v0, float *v1, float *v2, float *v3 ) {
    (*v0) = v_[0];
    (*v1) = v_[1];
    (*v2) = v_[2];
    (*v3) = v_[3];
  }

  float getMean( void ) {
    return mean_;
  }
  ...
};
...
```

Setter

Getter

Getter

Setters/getters may represent **additional cost** (more typing, function calls). Their use is usually justified when **getting/setting** a **data** member **involves additional consistency checks**!

All accesses to the **data** members are made through **setters** and **getters**.
Is it necessary for `setMean()` to be publicly accessible?

Setters / Mutators, Getters / Accessors

vector.cpp

```
class Vector {
public:
   ...
private:
   float v_[4];
   float mean_;

   void setMean( void ) {
     mean_ = (v_[0] + v_[1] + v_[2] + v_[3]) * 0.25f;
   }

};
...
};
...
```

Now **`setMean()`** is also **`private`**, i.e, it can **only** be **called** from **member functions** of the class!

Some Code Fine Tunning

vector.cpp

```
class Vector {
public:
  ...
  void copyFrom( const Vector *a ) {
    v_[0] = a->v_[0];
    v_[1] = a->v_[1];
    v_[2] = a->v_[2];
    v_[3] = a->v_[3];
    mean_ = a->mean_;
  }

  void sum( const Vector *a ) {
    v_[0] += a->v_[0];
    v_[1] += a->v_[1];
    v_[2] += a->v_[2];
    v_[3] += a->v_[3];
    mean_ += a->mean_;
  }
  ...
};
...
```

Pointer to `const`:

- Reduces the amount of data to be copied during the function call;
- Avoids the modification of the data pointed at.

Hey! How can we access **`a->v_[...]`** directly if **`v_`** is **private**?

Because, in **c++**, **access control** works on **per-class basis**, and not on per object basis! This feature is usually used when overloading operators.

Initialization of Data Members

- Traditionally, data members could receive an initial value in two different ways:
  - Member assignment.
  - Member initializer list.
- In C++11, data members can be initialized directly at their declaration.

Member Assignment

This is the method that we have used to "initialize" the data member of the class Vector:

vector.cpp

```
class Vector {
public:
    Vector( void )
    {
        v_[0]=0.0f;
        v_[1]=0.0f;
        v_[2]=0.0f;
        v_[3]=0.0f;
        mean_ = 0.0f;
    };
    ...
};
...
```

The data member `v_` receives its initial value through an assignment in the constructor

Isn't it Ok this way?

The main problem of **initializing** data members with an **assignment** in the constructor is that **it implies two operations**:

- **default constructor call**
- **assigment operator**

Member Initializer List

vector.cpp (old)

```
class Vector {
public:
    Vector( void )
    {
        v_[0]=0.0f;
        v_[1]=0.0f;
        v_[2]=0.0f;
        v_[3]=0.0f;
        mean_ = 0.0f;
    };
    ...
};
...
```

vector.cpp (new)

```
class Vector {
public:
    Vector( void ) :
        v_[0]( 0.0f ),
        v_[1]( 0.0f ),
        v_[2]( 0.0f ),
        v_[3]( 0.0f ),
        Mean_( 0.0f )
    { };
    ...
};
...
```

Initialization through **initializer list** is usually **faster** because there is only the cost of the **constructor**!

However, **neither of the two** (initializer list / assignment) is **natural**.
Is there an **alternative** (more natural) data member **initialization method**?

C++11 non-static Data Member Init.

It is also called “in place” initialization.

vector.cpp (old)

```
class Vector {
public:
    Vector( void ) :
        v_[0]( 0.0f ),
        v_[1]( 0.0f ),
        v_[2]( 0.0f ),
        v_[3]( 0.0f ),
        mean_( 0.0f )
    { };
    ...
};
...
```

vector.cpp (new)

```
class Vector {
public:
    Vector( void ) { };
    ...

    float v_[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
    float mean_ = 0.0f;
    ...
};
...
```

`const` and C++

- As in C, in C++ we may have:
  - `const` variables.
  - `const` pointers (even to `const` variables).
  - `const` function parameters (including pointer variations).
- But we might also have:
  - `const` data members.
  - `const` member functions.
  - `const` references.
  - `const` iterators.
  - Among others..

Let's take a look at some of these...

`const` Member Functions

Remember our *4-float-vector-plus-mean* class?

vector.cpp (old)

```
class Vector {
public:
  Vector( void ) { ... }        -> Alters class data.

  ~Vector( void ) {}            -> Might alter class data.

  void copyFrom( ... )          -> Alters class data.

  void sum( ... )               -> Alters class data.

  void setValues( ... }         -> Alters class data.

  void getValues( ... }         -> DOES NOT alter class data.

  float getMean( void )         -> DOES NOT alter class data.

private:
  void setMean( void )          -> Alters class data.
  ...
};
...
```

vector.cpp (new)

```
class Vector {
public:
  Vector( void ) { ... }

  ~Vector( void ) {}

  void copyFrom( ... )

  void sum( ... )

  void setValues( ... }

  void getValues( ... } const

  float getMean( void ) const

private:
  void setMean( void )
  ...
};
...
```

`const` member function indicates that the function **will not alter class internal data**.

const Member Functions

Back to our class simulation in C:

C++ class example.

class_cpp.cpp

```
class CPPClass {
public:
  int x_;
  int y_;

  void set( int a, int b ) {
    this->x_ = a;
    this->y_ = b;
  }

  int getX( void ) const {
    return this->x_;
  }
};

CPPClass cppobj;

int main( void )
{
  cppobj.set( 10, 20 );
  return cppobj.getX();
}
```

A **const** member function cannot alter its class data. It works as the **this** pointer was a pointer to **const**.

C class simulation.

class_c.c

```
struct CClass {
  int x_;
  int y_;
};

void Set( struct CClass *this,
          int a, int b ) {
  this->x_ = a;
  this->y_ = b;
}

int GetX( const struct CClass *this ) {
  return this->x_;
}

struct CClass cobj;

int main( void ) {
  Set( &cobj, 10, 20 );
  return GetX( &cobj );
}
```

`const` Data Members

Let's use our Vector class to build something else:

vector.cpp

```
class Vector {
    ...
};

class Color {
public:
    Vector rgba_;
};

int main( void ) {

    Color c;

    c.rgba_.setValues( 1.0f, 0.0f, 0.0f, 1.0f );

    return c.rgba_.getMean();
}
```

`rgba_` is a public data member of Color class.

What if, for some some reason, **`rgba_`** was a **`const` data member**?

`const` Data Members

vector.cpp (old)

```
class Vector {
    ...
};

class Color {
public:
    Vector rgba_;
};

int main( void ) {

    Color c;

    c.rgba_.setValues( ... );

    return c.rgba_.getMean();
}
```

We **cannot** invoke, for a const data member, **member functions** that try to **alter its data members**!

vector.cpp (new)

```
class Vector {
    ...
};

class Color {
public:
    const Vector rgba_;
};

int main( void ) {

    Color c;

    c.rgba_.setValues( ... );

    return c.rgba_.getMean();
}
```

Make an experiment: try removing the **`const`** qualifier from the definition of **`getMean()`**.

Try to compile this code...

What happened? How to solve that?

mutable

- Permits **data** members, declared within **`const`** member **functions**, to be **modified**.
- **Cannot** be applied to **`const`**, **`static`** names or **references**.

mutable.cpp

```
class X {
    mutable const int* p;
    mutable int* const q;
};
```

Ok
Not Ok

C++ References

- In the past (prior to C++11) a **reference** was meant to be a sort of ***alias*** to a **variable**.

mainref.cpp

```
int main( void )
{
    int i = 10;
    int &ri = i;

    return ri;
}
```

- From the C++11 on, **references** started to be considered as **variables**.

Pointers vs. References

- It is **not** possible to refer **directly** to a **reference object after** it is **defined**;
  - Any occurrence of its name refers directly to the object it references.
- Once a reference is **created**, it **cannot** be later made to **reference another** object;
- References **cannot be null**, whereas pointers can;
- References **cannot be uninitialized**;
- References which **are data members** of class instances **must be initialized** in the **initializer list** of the class's **constructor**.

References Implementation

- C++ spec does not describe **how references** should be **implemented**.
- However, **some compilers** implement C++ references in a way similar to **`const` pointers**.

References as `const` Pointers

mainref.cpp

```
int main( void ) {
    int i = 10;
    int &ri = i;

    return ri;
}
```

mainref.s

```
    .file   "mainref.cpp"
    .text
    .globl main
    .type   main, @function
main:
    pushq   %rbp
    movq    %rsp, %rbp
  ----------------------------
    movl    $10, -12(%rbp)
  ----------------------------
    leaq    -12(%rbp), %rax
    movq    %rax, -8(%rbp)
  ----------------------------
    movq    -8(%rbp), %rax
    movl    (%rax), %eax
    popq    %rbp
    ret
    ...
```

Move 10 to `i`, which is located at `-12(%rbp)`.

Copy the addr. of `i` to `-8(%rbp)`. Thus, `ri` is not just an alias to `i`, It actually occupies some memory space.

Copy the addr. of `i`, stored at `-8(%rbp)`, to `%rax` and dereference it to return the value of `i`.

Remember that, despite the **similarity** between the **assembly code**, it **does not** mean that **references are actually pointers**!

References as `const` Pointers

The example below illustrates the correspondence between references and **`const`** pointers:

mainref.cpp

```
int main( void ) {
    int i = 10;
    int &ri = i;

    return ri;
}
```

mainptr.cpp

```
int main( void ) {
    int i = 10;
    int * const ri = &i;

    return *ri;
}
```

Check out the assembly of the pointer based code!

Using References in the Vector Class

vector.cpp (old)

```
class Vector {
  ...
  void copyFrom( const Vector *a ) {
        v_[0] = a->v_[0];
        v_[1] = a->v_[1];
        v_[2] = a->v_[2];
        v_[3] = a->v_[3];
        mean_ = a->mean_;
  }
  void sum( const Vector *a ) {
        v_[0] += a->v_[0];
        v_[1] += a->v_[1];
        v_[2] += a->v_[2];
        v_[3] += a->v_[3];
        mean_ += a->mean_;
  }
  ...
  void getValues( float *v0,
                  float *v1,
                  float *v2,
                  float *v3 ) {
        (*v0) = v_[0];
        (*v1) = v_[1];
        (*v2) = v_[2];
        (*v3) = v_[3];
  }
  ...
```

vector.cpp (new)

```
class Vector {
  ...
  void copyFrom( const Vector &a ) {
        v_[0] = a.v_[0];
        v_[1] = a.v_[1];
        v_[2] = a.v_[2];
        v_[3] = a.v_[3];
        mean_ = a.mean_;
  }
  void sum( const Vector &a ) {
        v_[0] += a.v_[0];
        v_[1] += a.v_[1];
        v_[2] += a.v_[2];
        v_[3] += a.v_[3];
        mean_ += a.mean_;
  }
  ...
  void getValues( float &v0,
                  float &v1,
                  float &v2,
                  float &v3 ) {
        v0 = v_[0];
        v1 = v_[1];
        v2 = v_[2];
        v3 = v_[3];
  }
  ...
```

When to use pointers or references?

Scope Resolution Operator `::`

The scope resolution operator `::` "*is used to disambiguate identifiers used in different scopes*".

Microsoft MSDN

Scope Resolution Operator ::

Example

```
int a = 10;
namespace M
{
    int a = 20;
    namespace N
    {
        int a = 30;
        void f()
        {
            int x = a;
            int y = M::a;
            int z = ::a;
        }
    }
}
```

Global scope.

Visible within the `namespace` M.

Visible within the `namespace` N.

Uses the last declared `a`.

Qualified name: uses the `a` that was declared in the `namespace` M.

Qualified name: uses the global `a`.

Scope Resolution Operator ::

Member function definition example

class_cpp.cpp

```
class CPPClass {
public:
  int x_;
  int y_;

  void set( int a, int b ) {
    this->x_ = a;
    this->y_ = b;
  }

  int getX( void ) const {
    return this->x_;
  }
};

CPPClass cppobj;

int main( void ) {
  cppobj.set( 10, 20 );
  return cppobj.getX();
}
```

class_cpp.cpp

```
class CPPClass {
public:
    int x_;
    int y_;

    void set( int a, int b );
    int getX( void ) const;
};

void CPPClass::set( int a, int b ) {
    this->x_ = a;
    this->y_ = b;
}

int CPPClass::getX( void ) const {
    return this->x_;
}

CPPClass cppobj;

int main( void ) {
    cppobj.set( 10, 20 );
    return cppobj.getX();
}
```

`static` and C++

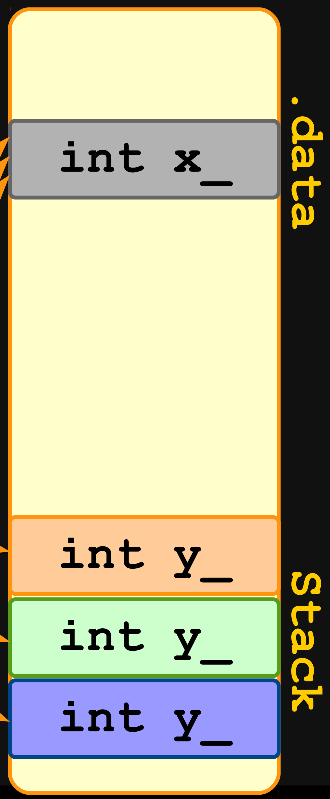
- In C `static` can be used with
  - Local variables.
  - Global variables.
- In C++ `static` can be used with
  - Data members.
  - Function members.

`static` Data Members

Similarly to what happened in C, in C++ `static` data members have **static duration**.

```
class Dummy {
public:
    static int x_;
    int y_;
};

int Dummy::x_ = 1;

int main( void ) {

    Dummy a;
    Dummy b;
    Dummy c;

    ...
```

Process virtual memory



Non-const `static` data members must be **initialized** (otherwise **symbols won't be** created)**!**

Static data members can be accessed through the scope resolution operator

```
int main( void ) {

    Dummy a;
    Dummy b;
    Dummy c;

    a.x_ = 10;
    b.x_ = 20;
    c.x_ = 30;

    // or

    Dummy::x_ = 40;

    ...
```

Refer to the same memory position.

static Data Members

Use example

Using a **static** data member to uniquely identify (number) each instance of a class.

object_id.cpp

```
class Dummy {
public:
    Dummy( void ) {
        setClassId();
    }

private:
    void setClassId( void ) {
        id_ = ++current_id_;
    }

    static int current_id_;
    int id_;
};

int Dummy::current_id_ = 0;

int main( void ) {
    Dummy a;
    Dummy b;

    return 0;
}
```

How about copies of the object? Think about that!

`static` Member Functions

It is **not required** that an instance of the class (*i.e.* object) exists so that we can invoke its `static` member functions.

`static` function members can **only** operate on **`static` data**!

object_id.cpp

```
#include <iostream>

class Dummy {
public:
    static void print( void ) {
        std::cout << "Hello world!" << std::endl;
    }
};

int main( void ) {

    Dummy::print();

    return 0;
}
```

Static function member

`print()` member function can be called, despite the fact that there is no instance of the class Dummy

`static` and the `this` Pointer

- In current C++ implementations, a **member function** works like an ordinary **C function** which receives an extra, **implicit**, parameter called **`this`** (which is a pointer to the current class instance that invokes the function).
- Also, **data members** are accessed implicitly through the **`this`** pointer (or implicitly, when **`this`** is not explicitly informed).

How a **`static`** member function handles the **`this`** pointer if it can be invoked **even if no class instance exists**, *i.e.*, even when **there is no `this` pointer**?

`static` and the `this` Pointer

Static member functions are, actually, even more like ordinary C functions, since they **do not** receive **`this`** as an extra, or implicit, **parameter**.

C++ classes.

Corresponding C class simulations.

nonstatic.cpp

```
class CPPClass {
public:
  int x_;

  void set( int a ) {
    this->x_ = a;
  }
  ...
```

nonstatic.c

```
struct CClass {
  int x_;
};

void Set( struct CClass *this, int a ) {
  this->x_ = a;
}
...
```

static.cpp

```
class CPPClass {
public:
  static int x_;

  static void set( int a ) {
    x_ = a;
  }
  ...
```

static.c

```
struct CClass {
  int x_;
} Var;

void Set( int a ) {
  Var.x_ = a;
}
...
```

As a side effect, **there is no** such a thing as a **`static const` member function**!

What about the **assembly** code **generated** for each program?

`static` functions vs. `namespace`

A **`namespace`** *is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.*

Microsoft MSDN

`static` functions vs. `namespace`

A **class** that contains **only `static` elements** (data and functions) behaves almost **like** a **`namespace`**.

class_namsp.cpp

```
...
class DummyClass {
public:
    static int x_;
    static void setX( int a );
};

void DummyClass::setX( int a ) {
    x_ = a;
}
...
```

class_namsp.cpp

```
...
namespace DummyNamespace {
    int x;
    void setX( int a );
}

void DummyNamespace::setX( int a ) {
    x = a;
}
...
```

`static` data member must be initialized

class_namsp.cpp

```
...

int DummyClass::x_ = 1;

int main( void )
{
    DummyClass::setX( 10 );

    DummyNamespace::setX( 20  );

    return 0;
}
```

Function Overload

- C++ allows one to **define multiple functions** with the **same identifier**, since their **parameter list differs** from each other.
- The **return** type **does not differentiate** functions with same identifier and parameter list.
- This feature is called **function overload**.

Function Overload

fover.cpp

```
#include <iostream>

void f( void ) {
    std::cout << "f void...." << std::endl;
}

void f( int x ) {
    std::cout << "f void...." << x << std::endl;
}

int main( void ) {

    f();

    f( 10 );

    return 0;
}
```

The function identifier `f` was overloaded.

fover.s

```
    ...
    .text
    .globl   _Z1fv
    .type    _Z1fv, @function
_Z1fv:
    pushq    %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
    ...
    .globl   _Z1fi
    .type    _Z1fi, @function
_Z1fi:
    pushq    %rbp
    movq %rsp, %rbp
    ...
    ret
    ...
main:
    ...
```

Which **address labels** the compiler generates for `f`?

Constructors

- **Constructor** is a special kind of **member function** that is responsible for the **initialization** of the **instances** (objects) of its class.
- Characteristics of the constructor include:
  - Has the **same name** as its **class**.
  - May be **overloaded**.
  - **Does not return** value.
  - If a constructor **is not provided** by the programmer, the compiler provides a **default constructor**.
  - Etc.

Constructors

vector.cpp

```
class Vector {
public:
1  Vector( void ) {};
2  Vector( float a ) :
       v_{ a, a, a, a },
       mean_ = a
   {};
3  Vector( float a, float b, float c, float d ) :
       v_{ a, b, c, d },
       mean_ = ( a + b + c + d ) * 0.25
   {};
4  Vector( const Vector &a ) :
       v_{ a.v_[0],  a.v_[1],  a.v_[2],  a.v_[3] },
       mean_ ( a.mean_ )
   {};

    float v_[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
    float mean_ = 0.0f;
};
...
```

1: Default constructor (explicitly informed).

2: Custom constructor.

3: Custom constructor.

4: Custom constructor.

Constructor invocation examples:

vector.cpp

```
Vector obj; 1

Vector obj( 25 ); 2

Vector obj( 1, 2, 3, 4 ); 3

Vector a( 25 ); 2
Vector obj( a ); 4

Vector obj(); !
```

MVP : Most Vexing Parse!

Solution:
C++11 uniform initialization

```
Vector obj{};
```

Copies

- Copy is an important operation in C++. For instance, the following operations involve copies:
  - Assignments
  - Parameter passing.
  - Etc.
- Example

How many **copies** of objects of class **Dummy** are actually being **made**?

obj_copies.cpp

```
class Dummy { ... }

Dummy f( Dummy x ) {
    Dummy y = x;     2º → Assignment.

    return y;        3º → Return by value.
}

int main( void ) {
    Dummy a;
                     4º → Assignment of a temporary object.
    Dummy b = f( a ); 1º → Parameter passing (by value).

    return 0;
}
```

Copy Constructors

- **New** objects, that are created as **copies** of **existing** objects, can do it through a call to a **copy constructor**.
- If a **copy constructor** is not explicitly provided by the programmer, a **default copy constructor** is automatically provided by the **compiler**.

Default Copy Constructor

Example

def_cpy_ctor.cpp

```
class Dummy {
public:
    Dummy( void ) {}

    Dummy( float a, int x, int y, int z ) :
        x_( a ),
        y_{ x, y, z } {}

    float x_ = 10.0f;
    int y_[3] = { 1, 2, 3 };
}

int main( void ) {
    Dummy a{ 11.0f, 11, 22, 33 };
    Dummy b = a; // equivalent to b( a )
    return 0;
}
```

```
Dummy( const Dummy &a ) :
    x_( a.x_ ),
    y_{ a.y_[0], a.y_[1], a.y_[2] }
{}
```

The default constructor, provided by the compiler, makes a bitwise copy of the source object into the destination object. Its effect is equivalent to:

```
std::memcpy(&b, &a, sizeof(Dummy));
```

Destination **Source** **Amount of bytes to be copied.**

This is also known as a **Shallow Copy!**

Default Copy Constructor

Back to our initial example:

obj_copies.cpp

```
class Dummy { ... }

Dummy f( Dummy x ) {
    Dummy y = x;        2º
    return y;           3º
}

int main( void ) {
    Dummy a;
                        4º
    Dummy b = f( a );   1º

    return 0;
}
```

- 2º → **Assignment.**
- 3º → **Return by value.**
- 4º → **Assignment of a temporary object.**
- 1º → **Parameter passing (by value).**

All operations above **may** involve **copies**!

Which operations are **actually** invoking the **default copy constructor**?

Default Copy Constructor

Modifying the initial code

obj_copies.cpp (old)

```
class Dummy { ... }

Dummy f( Dummy x ) {
    Dummy y = x;

    return y;
}

int main( void ) {
    Dummy a;

    Dummy b = f( a );

    return 0;
}
```

obj_copies.cpp (new)

```
class Dummy { ... }

Dummy f( Dummy &x ) {
    Dummy y;
    y = x;
    return y;
}

int main( void ) {
    Dummy a;

    Dummy b = f( a );

    return 0;
}
```

3º

4º

Now, **how many** times the **default copy constructor** will be invoked?

How about the other eventual **copy operations**?

Return by value.

Assignment of a temporary object.

Return Value Optimization!

Return Value Optimization (RVO)

original.cpp

Original code

```
struct Data {
  char bytes[16];
};

Data f() {
  Data result = {};
  // generate result
  return result;
}

int main() {
  Data d = f();
}
```

rvo.cpp

Actual code, with RVO.

```
struct Data {
  char bytes[16];
};

void f(Data *p) {
  // generate result directly in *p
}

int main() {
  Data d;
  f(&d);
}
```

No copy

copy.cpp

Actual code, without RVO.

```
struct Data {
  char bytes[16];
};

Data * f(Data * _hiddenAddress) {
  Data result = {};
  // copy res into the hidden obj
  *_hiddenAddress = result;
  return _hiddenAddress;
}

int main() {
  Data _hidden;
  Data d = * f(&_hidden);
}
```

Assignment

Copy

Limitations of Shallow Copies

Example

What happened here?

How to solve that?

Solution:
Deep Copy!

```
Dummy( const Dummy &a )
{
    size_ = a.size_;
    d_ = new int[size_];
    for ( int i = 0; i < size_; i++ )
        d_[i] = a.d_[i];
    }
```

shallowcpy.cpp

```
class Dummy {
public:
    Dummy( void ) {}

    Dummy( int size, int val ) :
        size_( size ) {
        d_ = new int[size_];
        for ( int i = 0; i < size_; i++ )
            d_[i] = val;
    }

    ~Dummy( void ) {
        if ( d_ )
            delete [] d_;
    }

    int *d_ = NULL;
    int size_ = 0;
}

int main( void ) {
    Dummy a{ 6, 888 };
    Dummy b = a;
    return 0;
}
```

References

- Classes I (discussion about constructors).
  - http://www.cplusplus.com/doc/tutorial/classes/
- An Insight to References in C++. itsdkg.
  - http://www.codeproject.com/Articles/13363/An-Insight-to-References-in-C
- Distinguish between pointers and references in C++
  - http://www.codeproject.com/Tips/91227/Distinguish-between-pointers-and-references-in-C