C++ Primer for 50.017

Why join this Course?

- Learn C++?
  - This short lesson won’t make you a C++ expert.
  - Neither will this course be sufficient.
- Learn OpenGL?
- Learn more algorithms?
- Become a better programmer?
- This course is like a all-you-can-eat buffet.
  The more you try, the more you get out of it.

Choose Your Weapon.

python™

- **Easy.**
- Beginner friendly.
- Many Libraries.
- Fast development.
- "Glue" language.
- Limited speed and control over low-level details.

Java™

- **Intermediate.**
- Beginner & Expert Friendly
- Many Libraries.
- Good for big projects / collaborations.

C++

- **Hard.**
- Expert friendly.
- Many Libraries.
- For high-performance code.
- Superior speed and control over low-level details.

Why C++?

- One of the fastest languages.
  http://codegolf.stackexchange.com/questions/26323/how-slow-is-python-really-or-how-fast-is-your-language

  Very important for lag-free graphics!
- **Both** high-level ***and*** low-level.
- Many graphical software and libraries.
- Very flexible (for experts).
- Requires a certain level of maturity to appreciate.

C++

- General-purpose programming language
- “C with classes”
- Developed in 1970’s by Bjarne Stroustrup
- Multi-paradigm
  - Object-oriented
  - Functional
  - Procedural
  - Generic

Main Areas of C++

- C Language Syntax
- Preprocessor
- Classes and Objects
- C++ Templates
- STL & Standard Library

Hello World

```cpp
#include <iostream>

int main(int argc, const char * argv[])
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Variables

```
// An integer
int a = -1;

// A non-negative integer
unsigned int b = 1;

// A real number (single-precision)
float d = 1.23f;

// A real number (double-precision)
double e = 2.71828182845904523536028747135227;

// A std::string (not a primitive type)
std::string s = "Hi, I am a string!";
```

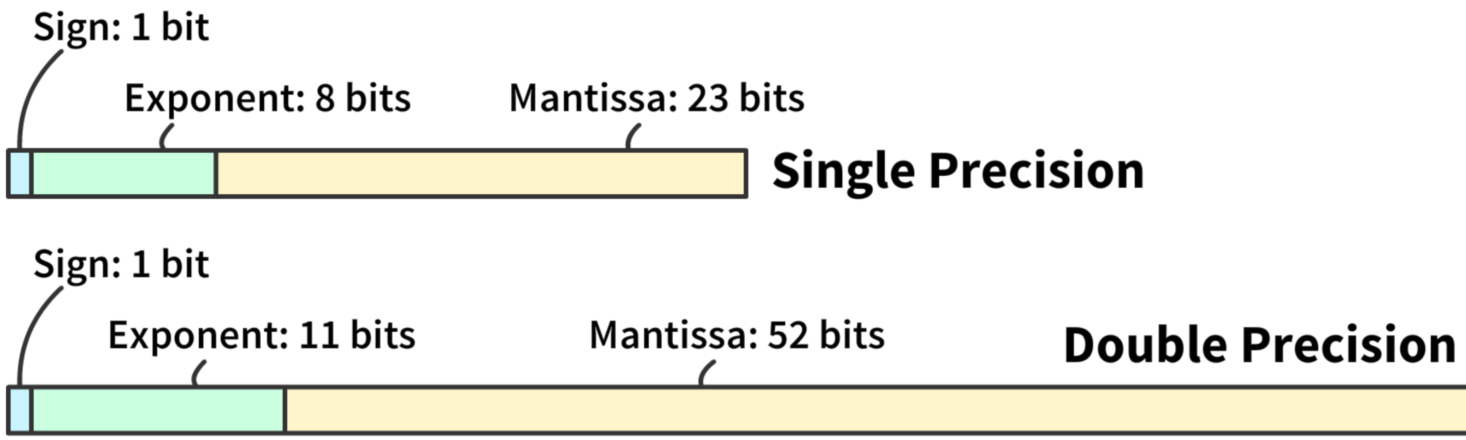
Integer Types

Type	Default Sign	Bits	Range
`bool`	Always unsigned	1	0 .. 1 false / true
`char`	Unsigned	8	Unsigned: 0 .. 255 Signed: -128 .. 127
`short`	Signed	16	Unsigned: 0 .. 65535 Signed: -32768 .. 32767
`int`	Signed	32	Unsigned: 0 .. 4294967295 Signed: -2147483648 .. 2147483647
`long`	Signed	32 (windows, 32-bit systems) 64 (64-bit unix, 64-bit osx)	Unsigned: 0 .. 2^{nBits} Signed: $-2^{nBits-1}$ .. $2^{nBits-1}-1$
`long long`	Signed	64	Unsigned: 0 .. 2^{64} Signed: -2^{63} .. $2^{63}-1$
`size_t`	Always unsigned	Depends on platform. Usually either 32 or 64.	Unsigned: 0 .. 2^{nBits} Signed: $-2^{nBits-1}$ .. $2^{nBits-1}-1$

You can prefix **`unsigned`** or **`signed`** to specify the sign. e.g. **`unsigned int`**

Floating Point Types

Type	IEEE745 Name	Bits	Decimal (Base 10) Precision	Range
`float`	Single precision	32	6-9 digits	-3.4E38 .. 3.4E38
`double`	Double precision	64	15-17 digits	-1.7E308 .. 1.7E308



$$\text{value} = (-1)^{\text{sign}} \times (1.\text{Mantissa}) \times 2^{\text{Exponent} - 127}$$

Valid Names

- ISO Latin Alphabet

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

- Western Arabic Numerals

```
0123456789
```

- ASCII Underscore

```
_
```

Reserved Keywords

You cannot use these words as names in C++.

alignas	const	friend	public	try
alignof	constexpr	goto	register	typedef
and	const_cast	if	reinterpret_cast	typeid
and_eq	continue	inline	requires	typename
asm	decltype	int	return	union
auto	default	long	short	unsigned
bitand	delete	mutable	signed	using
bitor	do	namespace	sizeof	virtual
bool	double	new	static	void
break	dynamic_cast	noexcept	static_assert	volatile
case	else	not	static_cast	wchar_t
catch	enum	not_eq	struct	while
char	explicit	nullptr	switch	xor
char16_t	export	operator	template	xor_eq
char32_t	extern	or	this	
class	false	or_eq	thread_local	
compl	float	private	throw	
concept	for	protected	true	

If / Else If / Else

```
if (i == 0) {
    std::cout << "i is zero. \n";
} else if (i == 1) {
    std::cout << "i is one. \n";
} else {
    std::cout << "i is something else. \n";
}
```

Switch

```
switch (i) {
    case 0:
        std::cout << "i is zero. \n";
        break;
    case 1:
        std::cout << "i is one. \n";
        break;
    default:
        std::cout << "i is something else. \n";
}

switch (i) {
    case 0:
    case 1:
    case 2:
        std::cout << "i is zero, one or two. \n";
        break;
    default:
        std::cout << "i is something else. \n";
}
```

Loops

For loops:

```
// Prints 0 .. 9
for (int i = 0; i < 10; i++) {
    std::cout << i << "\n";
}
```

While loops:

```
// Prints 0 .. 9
int i = 0;
while (i < 10) {
    std::cout << i << "\n";
    i++;
}
```

Loops

Breaking out of loops:

```
int i = 0;

// Prints 0 .. 9
while (true) {
    std::cout << i << "\n";
    i++;
    if (i == 10) break;
}
```

Continue. Skipping the rest of the loop's body:

```
int i = 0;

// Prints even numbers:
// 2, 4, 6, 8
while (i < 10) {
    if (i & 1) {
        i++;
        continue;
    }
    std::cout << i << "\n";
    i++;
}
```

Post vs Pre-increment

```
int i = 0;
std::cout << i++ << "\n"; // Prints 0
std::cout << i++ << "\n"; // Prints 1
std::cout << i++ << "\n"; // Prints 2

i = 0;
std::cout << ++i << "\n"; // Prints 1
std::cout << ++i << "\n"; // Prints 2
std::cout << ++i << "\n"; // Prints 3
```

STL and the Standard Library

Strings

```cpp
#include <iostream>
#include <string>  // For string

int main(int argc, const char * argv[])
{
    std::string s = "abc";

    std::string sCopy = s; // sCopy is a copy of s

    if (s == "abc") std::cout << "Strings Match \n";

    s += "defg"; // Appends to end of s

    std::cout << s << "\n"; // Prints abcdefg

    // Iterate through the string char by char
    for (size_t i = 0; i < s.size(); ++i) std::cout << s[i] << "\n";

    return 0;
}
```

Strings

```cpp
#include <iostream>
#include <string>  // For string

int main(int argc, const char * argv[])
{
    std::string s = "012345679";
    std::cout << s.substr(2, 3) << "\n"; // (start, length) Prints 234

    // Replace all occurances of 'cat' with '-'
    std::string haystack    = "catdogcatdogcatcatkitty",
                needle      = "cat",
                replacement = "-";
    while (1) {
        size_t f = haystack.find(needle);
        if (f == std::string::npos) break;
        haystack.replace(f, needle.length(), replacement);
    }
    std::cout << haystack << "\n";  // -dog-dog--kitty

    return 0;
}
```

As you can see, C++ can really get quite complex for some basic tasks.

Stringstream

Converting between string and numbers

```cpp
#include <iostream>
#include <string>  // For string
#include <sstream> // For stringstream
#include <vector>
#include <algorithm> // For replace

int main(int argc, const char * argv[])
{
    std::string numberString = "0,1,2,3,4,5,6";
    std::replace(numberString.begin(), numberString.end(), ',', ' ');

    std::stringstream ss;
    std::vector<int> numbers;

    ss.str(numberString); // load into the stringstream
    int n = 0;
    while (ss >> n) numbers.push_back(n);
    ss.clear(); // clear the stringstream

    for (size_t i = 0; i < numbers.size(); ++i)
        std::cout << numbers[i] << "\n";

    return 0;
}
```

STL Vector

A dynamically resizable array.

```
#include <iostream>
#include <vector> // For STL vector

int main(int argc, const char * argv[])
{
    std::vector<int> squares;

    // Append the squares of 0..9
    for (int i = 0; i < 10; i++) squares.push_back(i * i);

    // Another way of iterating
    for (size_t index = 0; index < squares.size(); index++)
        std::cout << squares[index] << "\n";

    // Another way of iterating
    // (C++ 11 and later... not available in VS2010)
    for (auto i : squares) std::cout << i << "\n";

    return 0;
}
```

STL Vector

More stuff…

```
#include <iostream>
#include <vector> // For STL vector
#include <algorithm> // Sort, copy, find

// A function for sorting... gives descending order
bool wayToSort(int i, int j) { return i > j; }

int main(int argc, const char * argv[])
{
    // Declare STL vector with 10 slots
    std::vector<int> squares(10);

    // Append the squares of 0..9
    for (int i = 0; i < 10; ++i) squares[i] = i * i;

    // Resize the vector to size 20, assigning 8 to new empty slots
    squares.resize(20, 8);

    // Check if the number 8 exists within
    // the start to end of the vector
    if (std::find(squares.begin(), squares.end(), 8) != squares.end())
        std::cout << "Found the number 8! \n";

    // Sort the vector within its start to end
    std::sort(squares.begin(), squares.end(), wayToSort);
```

STL Vector

More stuff…

```
    // Declare another STL Vector with 10 1's
    std::vector<int> anotherVector(10, 1);

    // Insert all elements of squares to the end of anotherVector
    anotherVector.insert(anotherVector.end(), squares.begin(),
        squares.end());

    // Copies all elements of squares into anotherVector,
    // replacing elements from the 5th (0 indexed) slot onwards
    std::copy(squares.begin(), squares.end(), anotherVector.begin() + 5);

    // Copies anotherVector to squares
    squares = anotherVector;

    // Clear the vector... (set it's size to 0)
    squares.clear();

    return 0;
}
```

STL Vector – Bonus

- The C++ STL has a list (double-linked-list) too.
  It is O(1) for inserting at the beginning and end.
- But it is highly recommended to use vector for most purposes
  (e.g. append to the end, sequential access).
  Performance wise, vector can outperform list by an order of magnitude. Why?
- Cache-coherence, branch-prediction,
  auto-vectorization, amortized-cost-insertion.

STL Map a.k.a. Dictionary

```
#include <iostream>
#include <map>

int main(int argc, const char * argv[])
{
    std::map<int, int> map; // Declare STL map

    map[172] = 2;
    map[172] = 30;
    map[2]   = 101;
    map[999] = 1234567;

    map.erase(map.find(999)); // Delete 999 and its value from the map

    map.find(2)->second = 1000;  // Find 2 and set its value to 1000
    map.find(98)->second = 1000; // No effect, as 98 isn't a key

    //  Iterate through the keys and values: {2: 1000, 172: 30}
    for (auto iter = map.begin(); iter != map.end(); ++iter)
        std::cout << "key: "   << iter->first  << ", "
                  << "value: " << iter->second << "\n";

    std::cout << map[172] << "\n"; // Access an element directly. Prints: 30
    return 0;
}
```

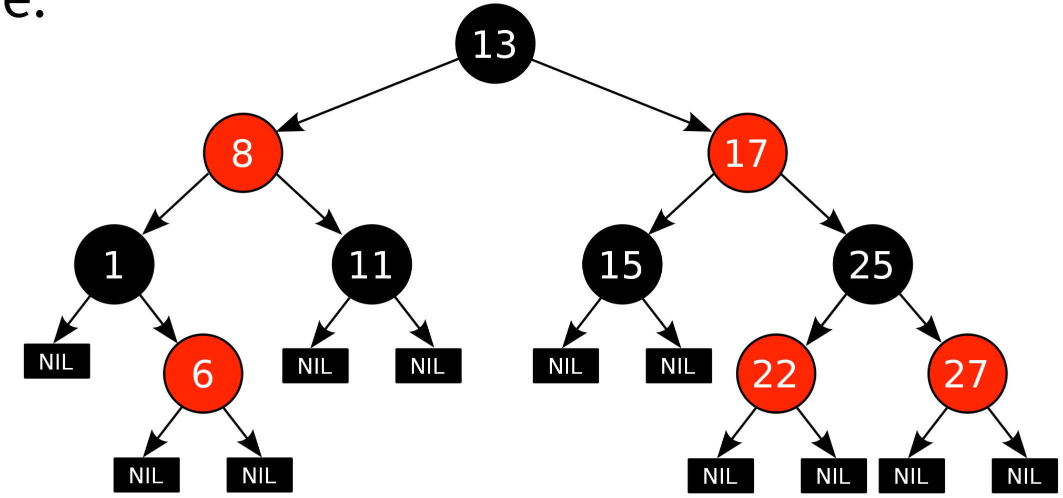
STL Set

A unique collection of items.

```cpp
#include <iostream>
#include <set> // For STL set

int main(int argc, const char * argv[])
{
    std::set<int> numbers; // Declare STL set

    numbers.insert(1);
    numbers.insert(2);
    numbers.insert(2);
    numbers.insert(3);
    numbers.insert(4);

    numbers.erase(numbers.find(4)); // Remove 4

    // Iterate through the set.. prints 1,2,3
    // The ternary expression checks if the iter is at the last element
    // to see if we should print a comma or line break.
    for (auto iter = numbers.begin(); iter != numbers.end(); ++iter)
        std::cout << *iter << (iter == --numbers.end() ? "\n" : ",");

    // Check if 3 exists in the set.
    if (numbers.find(3) != numbers.end()) std::cout << "3 Exists! \n";
    return 0;
}
```

STL Set / Map – Bonus!

- What data structure is used used for map / set?
  - Most C++ implementations use a Red-Black tree.
  - O(log n) time to insert, find, delete a key.
  - If you want O(1) performance, write your own!
  - Google Dense Hash (open-addressing)
  - std::HashMap, std::HashSet in C++ 11 (hash-buckets)



C++ File I/O

```cpp
#include <iostream>
#include <string>  // For string
#include <fstream> // For file i/o

int main(int argc, const char * argv[])
{
    // Write a file
    std::ofstream outFs("file.txt");
    if (outFs.is_open()) {
        for (int j = 0; j < 10; ++j) outFs << j << "\n";
    }
    outFs.close();

    // Read a file
    std::ifstream inFs("file.txt");
    std::string line;
    if (inFs.is_open()) {
        while (std::getline(inFs, line)) std::cout << line << "\n";
    }
    inFs.close();

    return 0;
}
```

Functions

Functions

```
void functionName(int argument0, int argument1)
{
    // do something
}

int add(int a, int b)
{
    return a + b;
}
```

Functions – Passing Arguments

```cpp
#include <iostream>

void passByValue(int a) { a++; }
void passByReference(int &a) { a++; }
void passByPointer(int *a) { (*a)++; }

int main(int argc, const char * argv[])
{
    int i = 0;

    passByValue(i);
    std::cout << i << "\n"; // 0.. no effect on i
    passByReference(i);
    std::cout << i << "\n"; // 1
    passByPointer(&i);
    std::cout << i << "\n"; // 2

    return 0;
}
```

Functions – Passing Arguments

For simple numeric types, (e.g. int, float, size_t), pass by value, unless you want to change them in the function.

For objects that may have a lot of data to be copied (e.g. stl containers), pass by reference.

For objects that may have a lot of data to be copied, and may be optionally included, pass by pointer.

```
void passByPointer(std::vector<int> *optional = NULL)
{
    if (optional != NULL) optional->push_back(1);
}
```

Functions – Passing Arguments

You can also use pass by reference or pass by pointer to “return” values from the function.

```
inline void HSVtoRGB(float  h, float  s, float  v,
                     float &r, float &g, float &b)
{
    if (s == 0) { r = g = b = v; return; }
    h /= 60;
    int i = floor(h);
    float f = h - i;
    float p = v * (1 - s);
    float q = v * (1 - s * f);
    float t = v * (1 - s * (1 - f));
    switch(i) {
        case 0:  r = v; g = t; b = p; break;
        case 1:  r = q; g = v; b = p; break;
        case 2:  r = p; g = v; b = t; break;
        case 3:  r = p; g = q; b = v; break;
        case 4:  r = t; g = p; b = v; break;
        default: r = v; g = p; b = q; break;
    }
}
```

Function Overloading

You can define functions of the same name that accept different arguments. The appropriate function will be called.

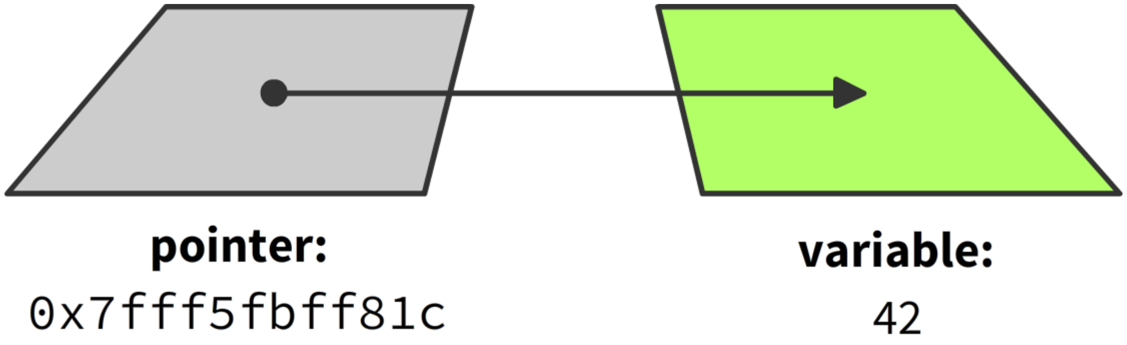
```cpp
#include <iostream>
#include <string>

int add(int v0, int v1) { return v0 + v1; }

int add(int v0, int v1, int v2) { return v0 + v1 + v2; }

std::string add(std::string s0, std::string s1)
{ return s0 + s1; }

int main(int argc, const char * argv[])
{
    std::cout << add(1, 2) << "\n"; // 3
    std::cout << add(1, 2, 3) << "\n"; // 6
    std::cout << add("one", "two") << "\n"; // onetwo

    return 0;
}
```

Pointers

Pointers

- *Explicit* Pointers are core to C++.
- A pointer stores the memory address to a variable.
- To denote a pointer for a **Type**, we put a ***** after the **Type**.
  (the whitespace before or after the * doesn't matter)

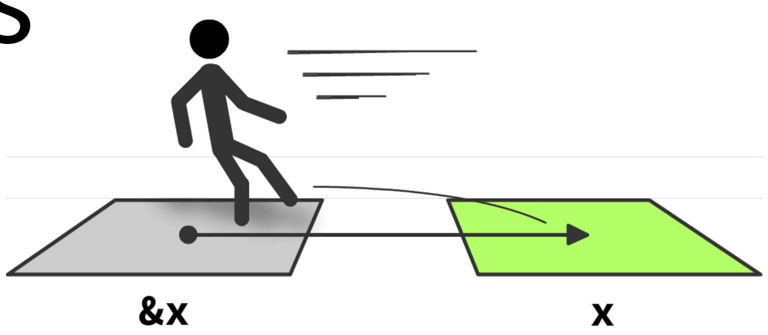
```
int variable = 42;
int *pointer = &variable;
```



Pointer Operations

&x

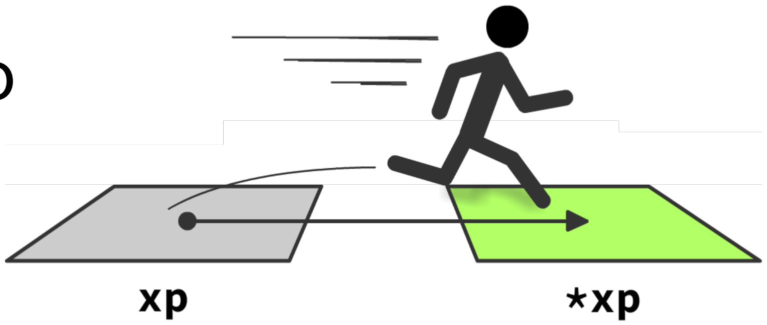
Address-of **x**
("Step backwards")



```
int i = 1;
std::cout << i << "\n"; // 1
std::cout << &i << "\n"; // Address of i
```

***xp**

Variable pointed to by **xp**
("Step forward")



```
int *ip = new int(2);
std::cout << ip << "\n"; // Address of the int
std::cout << *ip << "\n"; // 2
```

Manual Memory Management

```
Type *pointer = new Type(arguments);
```

Reserves memory for one Type object,
call’s the constructor on the object,
and assigns its address to pointer.

```
delete pointer;
```

Calls the destructor on the object,
and releases the memory.

Manual Memory Management

```
Type *array = new Type[N];
```

Reserves memory for **N** Type objects,
call’s the constructor on each object,
and assigns the starting memory address to the pointer.

```
delete [] array;
```

Calls the destructor on each object,
and releases the memory.

Manual Memory Management

```
Type *array = (Type *) malloc(N * sizeof(Type));
```

Reserves memory for **N** Type objects,
and assigns the starting memory address to the pointer.

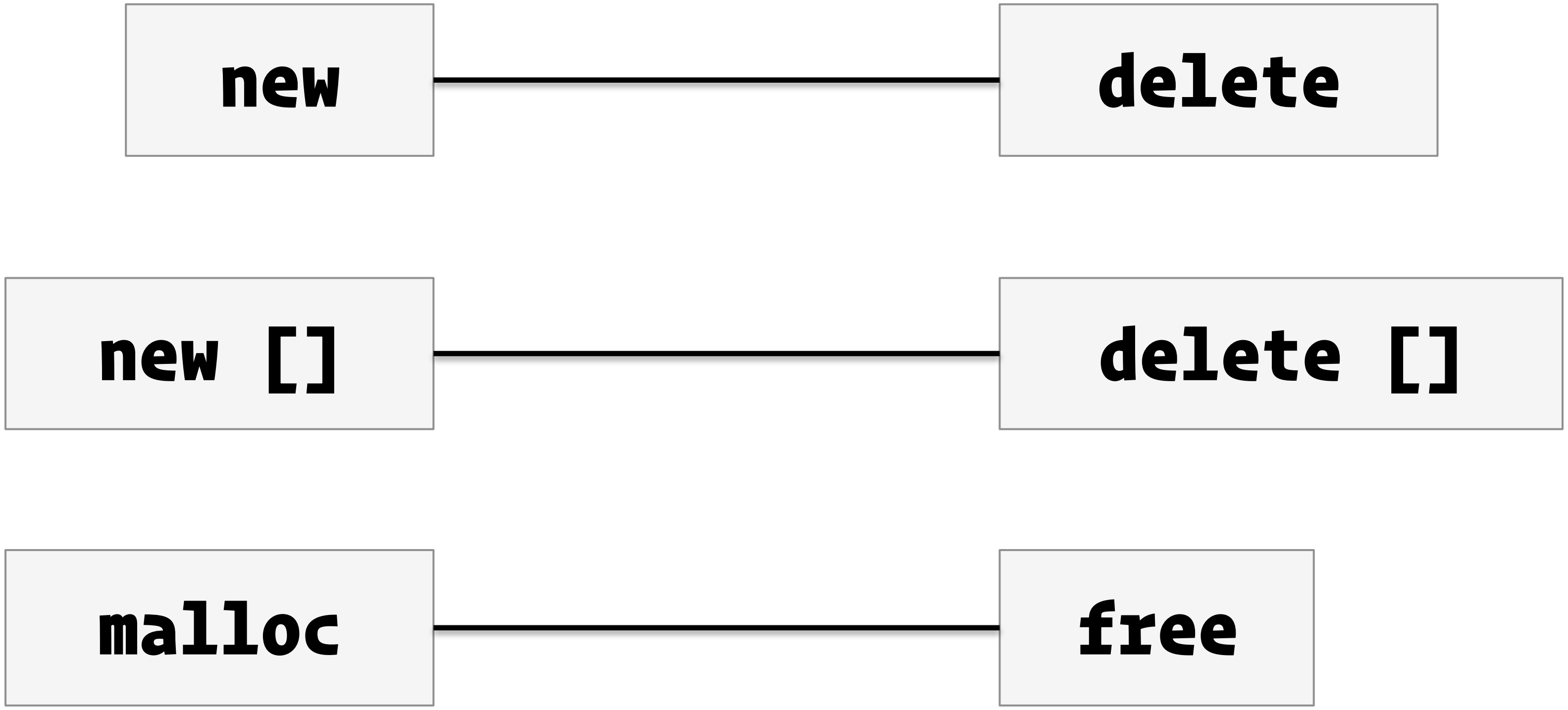
```
free(array);
```

Releases the memory.

malloc and **free** are found in stdlib.h,
which needs to be included before you use them:

```
#include <stdlib.h>
```

What to Use for Releasing Memory



Pointer Based Array

```
#include <iostream>

int main(int argc, const char * argv[])
{
    size_t *arr = new size_t[128];

    for (size_t i = 0; i < 128; ++i) arr[i] = i * 2;

    std::cout << arr           << "\n"; // 0x100801200
    std::cout << arr + 1       << "\n"; // 0x100801208
    std::cout << arr[1]        << "\n"; // 2
    std::cout << arr[10]       << "\n"; // 20
    std::cout << *(arr + 10) << "\n"; // 20

    delete [] arr;

    return 0;
}
```

Pointer Based 2D Array

```
#include <iostream>

int main(int argc, const char * argv[])
{
    // 1d array of 1d arrays
    size_t **arr = new size_t* [8];

    for (size_t i = 0; i < 8; ++i) {
        arr[i] = new size_t [8];
        for (size_t j = 0; j < 8; ++j)
            arr[i][j] = i * 8 + j;
    }

    std::cout << arr[7][7] << "\n"; // 7 * 8 + 7 = 63

    // You have to first delete each row individually.
    for (size_t i = 0; i < 8; ++i) delete [] arr[i];
    // Then delete arr.
    delete [] arr;

    return 0;
}
```

Manual Memory Management

The memory reserved with **new / malloc** will stay reserved throughout the lifetime of the program.

If you keep calling **new / malloc** without releasing the memory when it is no-longer required,
the program will hog up more and more memory without actually using it.

This can be an issue for long running programs
(e.g. Large Physical Simulations, Web Browsers, Games).

Memory Leak

```
int *ip = new int(2);      // Allocate memory
ip = new int(8);           // Allocate memory
```

The previously allocated memory is still reserved.

But since we have overridden **`ip`** with a new address,
we have *discarded* the previous address
(here, we have **ip** as the only copy of the address).

Without the address, we can’t find the previous
variable and cannot delete / free it to release memory.

Memory Leak - Fix

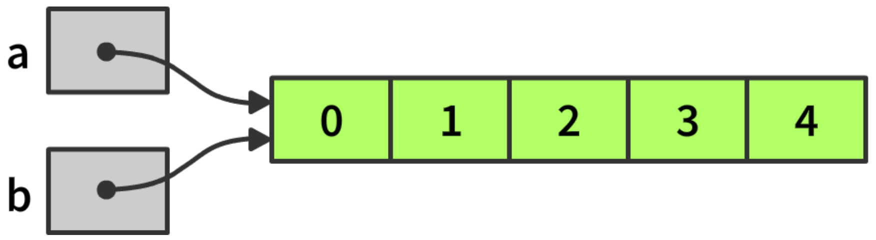
```
int *ip = new int(2); // Allocate memory
delete ip;            // Release the memory
ip = new int(8);      // Allocate memory
```

C++ does not have automatic garbage collection.

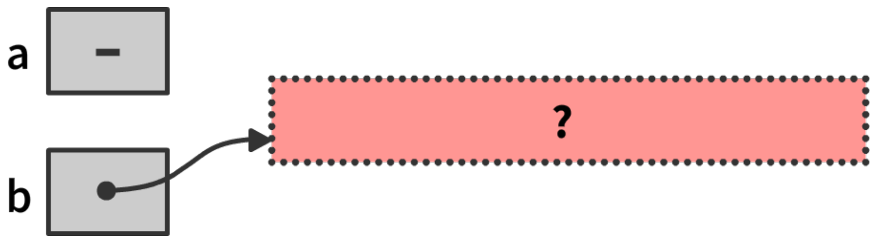
Make sure you manually release the memory before you discard the address.

Dangling Pointer

```
int *a = new int[5];
for(int i = 0; i < 5; ++i)
    a[i] = i;
int *b = a;
```



```
delete [] a;
b[0] = 1; // illegal!
```



Once **a** is deleted, **b** is still pointing to the same memory address.

This may cause a crash as the memory address pointed to may not belong to the program anymore!

Note: Creating dangling pointers *is not wrong* – sometimes you have to keep multiple pointers to the same address, and will thus create dangling pointers upon delete. Just be careful not to access deleted regions. Code carefully.

Some Advice

As you can see, manually managing memory can be difficult, tedious, and prone to errors.

It is recommended to avoid it.

Stick to the STL as far as possible. (std::vector, std::string, etc.)

The STL classes have been expertly designed many advanced C++ techniques to *auto-magically* implement all these memory management stuff under the hood.

For most use cases, the STL classes are as fast as manual memory management by an expert.

Classes and Objects

A Simple Class

```
class SimpleClass
{
    // unless marked public, members and functions are
    // private by default
    int i;

public:
    // If an argument has the same name as
    // a member, prefix 'this->' to differentiate
    // the class member from the argument.
    void setValue(int i) { this->i = i; }

    // The const keyword denotes that calling this function
    // won't mutate an instance of the class.
    // Required when passing an object as a const arugment
    // into a function
    int getValue() const { return i; }
};
```

Constructors

```cpp
class ABC
{
    int a, b, c;
public:
    ABC(int a, int b, int c)
    {
        this->a = a;
        this->b = b;
        this->c = c;
    }
};
```

```cpp
class ABC
{
    int a, b, c;
public:
    ABC(int a, int b, int c):
    a(a),
    b(b),
    c(c) {}
};
```

Both constructors are the same.
To make a new instance of ABC, simply call:

```cpp
ABC abc(1,2,3);
```

or

```cpp
ABC abc = ABC(1,2,3);
```

Default Constructors

```cpp
class ABC
{
    int a, b, c;
public:
    ABC(int a, int b, int c): a(a), b(b), c(c) {}
};

int main(int argc, const char * argv[])
{
    // Compile error:
    // No matching constructor for initialization of 'ABC'
    ABC *abcs = new ABC[10];
    delete [] abcs;
    return 0;
}
```

When you declare a custom constructor for a class, its default constructor is overwritten. This can cause a compile error when trying to declare an array of Objects which requires a default constructor.

Default Constructors – Fix

```
class ABC
{
    int a, b, c;
public:
    ABC(int a, int b, int c): a(a), b(b), c(c) {}
    ABC() {} // Add back the default constructor
};

int main(int argc, const char * argv[])
{
    ABC *abcs = new ABC[10];
    for (int i = 0; i < 10; ++i) abcs[i] = ABC(1,2,3);
    delete [] abcs;
    return 0;
}
```

The fix is to add back the default constructor.

After using new to create the array, do make sure to initialize the elements before accessing them if needed. (or else their members will be undefined.)

Destructors

(Used for manual memory management.)

```
#include <iostream>

struct Node
{
    Node *next;  int value;
    Node(int value, Node *next = NULL): value(value), next(next) {};

    ~Node() // Destructor
    {
        std::cout << value << " ";
        delete next;
    }
};

int main(int argc, const char * argv[])
{
    Node *n = new Node(0);
    n = new Node(1, n);  n = new Node(2, n);  n = new Node(3, n);
    delete n; // Prints: 3 2 1 0 as the destructors are called recursively
    return 0;
}
```

Subclassing

```cpp
class Object3D
{
protected:
    Vector4f color;
public:
    Object3D(const Vector4f &color): color(color) {}
    Object3D() {}
};

class Sphere: public Object3D
{
protected:
    float radius;
    Vector3f center;
public:
    Sphere(const Vector4f &color, const Vector3f &center, float radius):
    Object3D(color), // pass on arguments to the superclass constructor
    center(center),
    radius(radius) {}
};
```

Virtual Methods

(for Polymorphism)

```
class Object3D
{
protected:
    Vector4f color;
public:
    Object3D(const Vector4f &color):
    color(color) {}
    Object3D() {}

    virtual float area() { return 0; }
};

class Sphere: public Object3D
{
protected:
    float radius;
    Vector3f center;
public:
    Sphere(const Vector4f &color,
           const Vector3f &center,
           float radius):
    Object3D(color),
    center(center),
    radius(radius) {}

    float area()
    {
        return 4.0 * M_PI *
        radius * radius;
    }
};
```

```
class Triangle: public Object3D
{
protected:
    Vector3f v0, v1, v2;
public:
    Triangle(const Vector4f &color,
             const Vector3f &v0,
             const Vector3f &v1,
             const Vector3f &v2):
    Object3D(color), v0(v0), v1(v1), v2(v2) {}

    float area()
    {
        return Vector3f::cross(v2-v0, v1-v0).abs() * 0.5;
    }
};


int main( int argc, char* argv[] )
{
    Object3D *objects[2];

    objects[0] = new Sphere(
        Vector4f(1,0,0,1),
        Vector3f(1,1,1),
        2);

    objects[1] = new Triangle(
        Vector4f(1,0,0,1),
        Vector3f(0,0,1),
        Vector3f(1,1,1),
        Vector3f(2,0,1));

    // The respective area() will get called.
    for (int i = 0; i < 2; ++i)
        std::cout << objects[i]->area() << "\n";

    for (int i = 0; i < 2; ++i)
        delete objects[i];

    return 0;
}
```

Pure Virtual Methods

```
class Object3D
{
protected:
    Vector4f color;
public:
    Object3D(const Vector4f &color): color(color) {}
    Object3D() {}
    virtual float area() = 0;
};
```

To make a *pure virtual function*, we set a virtual function to 0.
It has a prototype, but no definition.

A class with a *pure virtual function* is called a *pure virtual class* and cannot be instantiated. (However, its subclasses can.)

‘.’ vs ‘->’

```
#include <iostream>

struct ABC
{
    // A struct is a class with its members public by default
    int a, b, c;
    ABC(int a, int b, int c):a(a), b(b), c(c) {}
    int getSum() { return a + b + c; }
};

int main(int argc, const char * argv[])
{
    ABC abc0(1,2,3);
    int sum0 = abc0.getSum();
    int a0 = abc0.a;

    ABC *abc1 = new ABC(1,2,3);
    int sum1 = abc1->getSum();
    int a1 = abc1->a;
    delete abc1;

    return 0;
}
```

We use the ‘.’ operator to access a member if the object is referred to directly.

We use the ‘->’ operator to access a member if the object is referred to by a pointer.

References

```
ABC abc(1,2,3);
ABC &abcRef = abc; // ABC is not copied
int &aRef = abc.a; // a is not copied

abcRef.a = 10;
std::cout << abc.a << " " << abc.b << " " << abc.c << "\n"; // 10 2 3

ABC *abcPtr = &abc;
abcPtr->a = 20;
std::cout << abc.a << " " << abc.b << " " << abc.c << "\n"; // 20 2 3

ABC abcCopy = abc;
abcCopy.a = 70;
std::cout << abc.a << " " << abc.b << " " << abc.c << "\n"; // 20 2 3

ABC thisIsAVeryLongVariableName(1,2,3);
ABC &t = thisIsAVeryLongVariableName;
```

When we make a reference to a variable, it is not copied.
Same like making a pointer to it.

Note that references must bind to a variable upon instantiation, while pointers can point to NULL to denote an absent value.

Other Stuff

Templates

```cpp
#include <iostream>

template <class T> void quickSort(T *a, size_t n)
{
    if (n < 2) return;
    T p = a[n / 2], t, *i, *j;
    for (i = a, j = a + n - 1; ; ++i, --j) {
        while (*i < p) ++i;
        while (p < *j) --j;
        if (i >= j) break;
        t = *i, *i = *j, *j = t;
    }
    quickSort(a, i - a);
    quickSort(i, n - (i - a));
}

int main(int argc, const char * argv[])
{
    double arr[10] =
    {7, 2, 2, 10, 5, 2, 10, 11, 1, 9};
    quickSort(arr, 10);
    for (size_t i = 0; i < 10; ++i)
        std::cout << arr[i] << " ";
    return 0;
}
```

Generic programming via templates are one of the most powerful features of C++.

Essentially, templates are compile time polymorphism.

When you call a function template, the compiler automatically detects the type for the template.

Then it creates a new function, just for the type, and use the function on the arguments.

Namespaces

```cpp
#include <iostream>

namespace myNamespace0
{
    struct S { int a, b; };
}

namespace myNamespace1
{
    struct S { int a, b, c; };
}

int main(int argc, const char * argv[])
{
    myNamespace0::S s0 = {1,2};
    myNamespace1::S s1 = {1,2,3};
    return 0;
}
```

Namespaces are useful for organizing code for large projects.

For example, you may have two different libraries that have a class ABC.

Namespaces can be use to differentiate between these two classes.

Namespaces “using”

```cpp
#include <iostream>

namespace myNamespace
{
    struct S { int a, b, c; };
}

int main(int argc, const char * argv[])
{
    myNamespace::S s = {1,2,3};
    return 0;
}
```

```cpp
#include <iostream>

namespace myNamespace
{
    struct S { int a, b, c; };
}

int main(int argc, const char * argv[])
{
    using namespace myNamespace;
    // can omitt myNamespace::
    S s = {1,2,3};
    return 0;
}
```

With `using namespace myNamespace;` we can omit prefixing the namespace.

A common way to avoid typing `std::` is to place `using namespace std;` at the relevant scope. (try not to place it in global scope)

Preprocessor Includes

```
#include <iostream>
```

The “`#include`” directive tells the preprocessor to copy the entire contents of the file and paste it in the spot.

```
#include "my_header.h"
```

When including files that are not in the system library, we use the double-quotes instead of angled brackets.

Useful C Functions

```
#include <stdio.h>
#include <string.h>

int main(int argc, const char * argv[])
{
    printf("%d, %f, %s\n", 1, 2.3, "abc");

    int *numbers = new int[10];
    memset(numbers, 0, sizeof(int) * 10);
    for (int i = 0; i < 10; ++i) numbers[i] = i;

    int *numbersCopy = new int[20];
    memcpy(numbersCopy, numbers, sizeof(int) * 10);
    memcpy(numbersCopy + 10, numbers, sizeof(int) * 10);
    for (int i = 0; i < 20; ++i) printf("%d ", numbersCopy[i]);

    delete [] numbers;
    delete [] numbersCopy;

    return 0;
}
```

Operator Overloading

Make your code more concise with syntactic sugar.

```
#include <iostream>
#include <stdio.h> // For sprintf

struct Vec3
{
    float x, y, z;
    Vec3(float x, float y, float z): x(x), y(y), z(z) {}
};

inline Vec3 operator + (const Vec3 &a, const Vec3 &b)
{
    return Vec3(a.x + b.x, a.y + b.y, a.z + b.z);
}

inline std::ostream & operator << (std::ostream& os, const Vec3 &v)
{
    char temp[64];
    sprintf(temp, "[ %.4f, %.4f, %.4f ]", v.x, v.y, v.z);
    return os << temp;
}

int main(int argc, const char * argv[])
{
    std::cout << Vec3(1,2,3) + Vec3(4,5,6) << "\n"; // [ 5.0000, 7.0000, 9.0000 ]
    return 0;
}
```

Functions Pointers

Passing around functions as variables. (old-school way)

```
#include <iostream>

void applyFunction(int *arr, size_t arrSize, int (*func)(int))
{
    for (size_t i = 0; i < arrSize; ++i)
        arr[i] = func(arr[i]);
}

int sq(int a) { return a * a; }

int main(int argc, const char * argv[])
{
    int arr[10] = {0,1,2,3,4,5,6,7,8,9};

    applyFunction(arr, 10, sq);
    for (size_t i = 0; i < 10; ++i)
        std::cout << arr[i] << "\n";

    return 0;
}
```

Functors

Combines 2 tricks:

- Using operator() to make an object act like a function.
- Using templates to allow passing in the function object.

This technique is heavily used in the STL.

It is also more versatile and gives better performance than function pointers.

```cpp
#include <iostream>

template <class Func>
void applyFunc(int *arr, size_t n, Func &f)
{
    for (size_t i = 0; i < n; ++i)
        arr[i] = f(arr[i]);
}

struct CumulativeSum
{
    int sum;

    inline int operator () (int a)
    { sum += a; return sum; }
};

int main(int argc, const char * argv[])
{
    int arr[10] = {0,1,2,3,4,5,6,7,8,9};

    CumulativeSum c;
    c.sum = 0;
    applyFunc(arr, 10, c);
    for (size_t i = 0; i < 10; ++i)
        std::cout << arr[i] << "\n";

    return 0;
}
```

File Organization

```cpp
#ifndef abc_hpp
#define abc_hpp

struct ABC
{
    int a, b, c;
    ABC(int a, int b, int c);
    int getSum();
};

#endif
```

abc.h

```cpp
#include "abc.h"

ABC::ABC(int a, int b, int c):
a(a), b(b), c(c) {}

int ABC::getSum()
{ return a + b + c; }
```

abc.cpp

```cpp
#include "abc.h"

int main(int argc, const char * argv[])
{
    ABC abc(1,2,3);
    return 0;
}
```

main.cpp

Header Only File Organization

```
#ifndef abc_hpp
#define abc_hpp

template<class T> struct ABC
{
    T a, b, c;
    inline ABC(T a, T b, T c):
    a(a), b(b), c(c) {}
    inline T getSum() { return a + b + c; }
};

#endif
```

abc.h

```
#include "abc.h"

int main(int argc, const char * argv[])
{
    ABC <int> abc(1,2,3);
    return 0;
}
```

main.cpp

Another popular method is to omitt the cpp file.

This is useful for template classes and for inline functions.

You will also need to type less.

The downside is longer compilation time. For moderately sized projects, this shouldn’t be an issue.

Preprocessor Macros

```
#include <iostream>
#define TWO 2
#define THREE 1 + 2

int main(int argc, const char * argv[])
{
    std::cout << TWO << "\n";             // 2
    std::cout << THREE << "\n";           // 3
    std::cout << THREE * TWO << "\n";     // 5
    std::cout << (THREE) * TWO << "\n";   // 6
    return 0;
}
```

Text substitutions before compilation.

It's essentially just copying and pasting code.

```
#include <iostream>
#define MAX(a, b) ((a)>(b)?(a):(b))

int main(int argc, const char * argv[])
{
    std::cout << MAX(1,2) << "\n"; // 2
    return 0;
}
```

Can be used to make "functions". Be careful of the parenthesis!

Can be to cut down verbosity and make your life easier.

Closing Notes

Interpreted vs Compiled

Interpreted languages

- Not converted into machine code.
- Syntax has to be parsed and understood during runtime.
- Examples: Python, Ruby, PHP, Java, C#

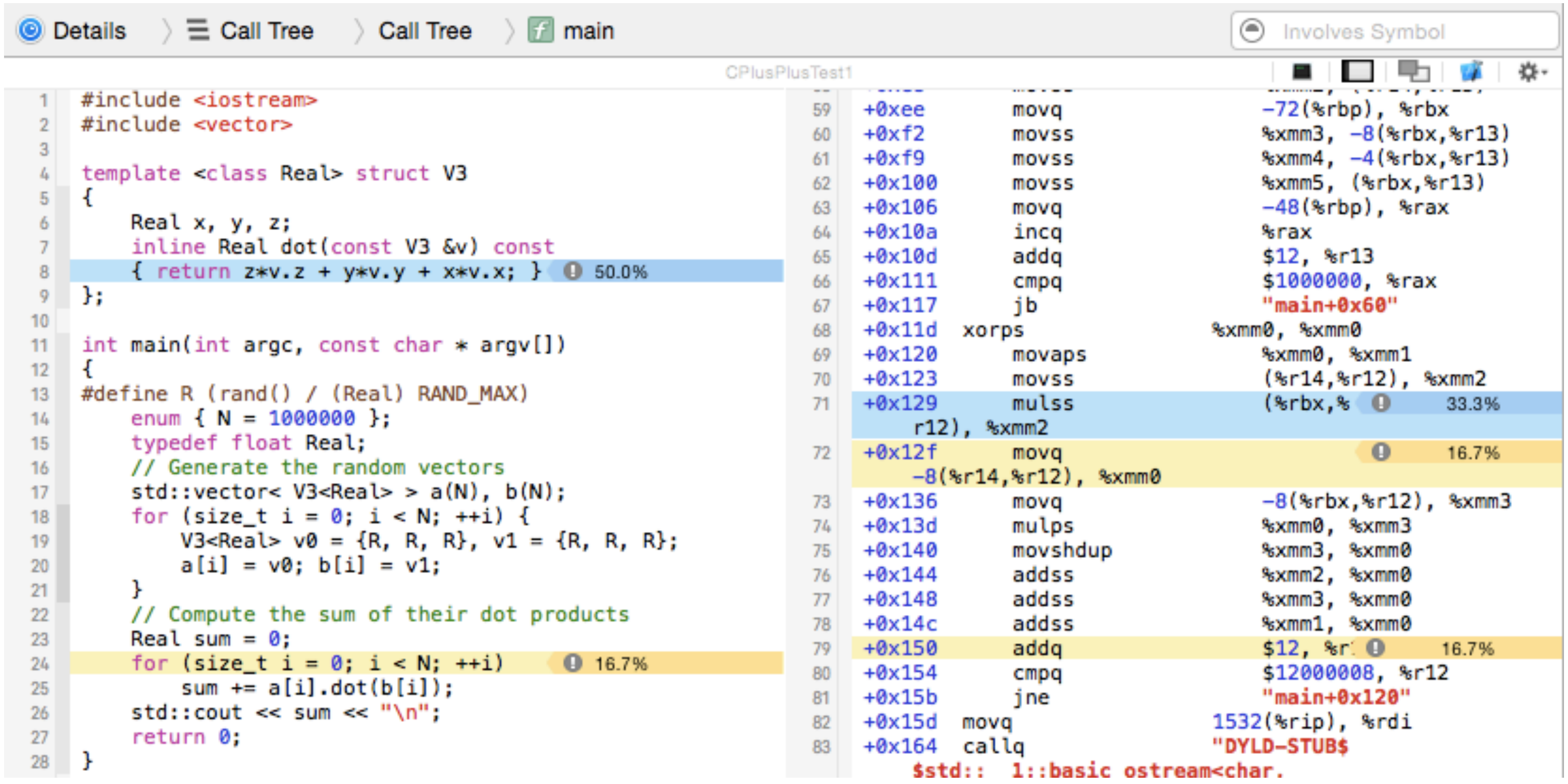
Compiled languages

- Converted into machine code.
- Machine code directly executed during runtime.
- Examples: C++, C, Objective-C, Swift, GLSL, Lisp, Fortran, Go

The Zero-overhead Principle

- This comes from C++ compilation process.
- You can build up lots of abstraction (classes, templates, etc). They are collapsed into plain machine code when compiled!
- Modern compilers can add many optimizations such as *inlining, loop-unrolling, constant-folding, copy-elision, auto-vectorization, etc.*
- After compilation, there is no notion of classes, templates, etc. Just plain machine instructions operating on registers…
  add, subtract, multiply, etc.
  It is as if the program is directly written in assembly!
- This is the source of C++'s speed!

The Zero-overhead Principle



```cpp
#include <iostream>
#include <vector>

template <class Real> struct V3
{
    Real x, y, z;
    inline Real dot(const V3 &v) const
    { return z*v.z + y*v.y + x*v.x; }   50.0%
};

int main(int argc, const char * argv[])
{
#define R (rand() / (Real) RAND_MAX)
    enum { N = 1000000 };
    typedef float Real;
    // Generate the random vectors
    std::vector< V3<Real> > a(N), b(N);
    for (size_t i = 0; i < N; ++i) {
        V3<Real> v0 = {R, R, R}, v1 = {R, R, R};
        a[i] = v0; b[i] = v1;
    }
    // Compute the sum of their dot products
    Real sum = 0;
    for (size_t i = 0; i < N; ++i)   16.7%
        sum += a[i].dot(b[i]);
    std::cout << sum << "\n";
    return 0;
}
```

```
+0xee       movq        -72(%rbp), %rbx
+0xf2       movss       %xmm3, -8(%rbx,%r13)
+0xf9       movss       %xmm4, -4(%rbx,%r13)
+0x100      movss       %xmm5, (%rbx,%r13)
+0x106      movq        -48(%rbp), %rax
+0x10a      incq        %rax
+0x10d      addq        $12, %r13
+0x111      cmpq        $1000000, %rax
+0x117      jb          "main+0x60"
+0x11d    xorps       %xmm0, %xmm0
+0x120      movaps      %xmm0, %xmm1
+0x123      movss       (%r14,%r12), %xmm2
+0x129      mulss       (%rbx,%   33.3%
    r12), %xmm2
+0x12f      movq                  16.7%
    -8(%r14,%r12), %xmm0
+0x136      movq        -8(%rbx,%r12), %xmm3
+0x13d      mulps       %xmm0, %xmm3
+0x140      movshdup    %xmm3, %xmm0
+0x144      addss       %xmm2, %xmm0
+0x148      addss       %xmm3, %xmm0
+0x14c      addss       %xmm1, %xmm0
+0x150      addq        $12, %r   16.7%
+0x154      cmpq        $12000008, %r12
+0x15b      jne         "main+0x120"
+0x15d    movq        1532(%rip), %rdi
+0x164    callq       "DYLD-STUB$
    $std::  1::basic ostream<char.
```

Once compiled, the dot product function is simply a sequence of instructions to load, multiply, add and store registers.

For Assignments

- Please make sure your code can compile and run on Microsoft Visual Studio 2010.
  Even if you write on OSX or some other environment.
- It supports C++ 98, and a subset of C++ 11.
- Learn to write cross-platform, backward-compatible code.
  Knowing what is supported in different environments is an essential C++ skill.

More Resources

- Add me on Facebook. (Benjamin Kang Yue Sheng) Contact me whenever you have a question!
- Stackoverflow
- Lynda
- Foundations of 3D Computer Graphics (Steven J. Gortler)
- 3D Computer Graphics – A Mathematical Introduction with OpenGL (Samuel R. Buss)
- Do your assignments and learn along the way