Basic Programming in C++

Santosh Ansumali[1]

[1]Engineering Mechanics Unit
Jawaharlal Nehru Centre for Advanced Scientific Studies
Bangalore, India

July 31, 2016

Overflow and Underflow

Overflow

Consider a unsigned int data type of 1 byte (range is 256):

$$200 + 100 = 44$$

What happened?? 200 in binary is 11001000 and 100 in binary is 01100100, 300 in binary is 100101100, which is more than the max value 11111111. So the higher byte will be rejected and result is 44. This is an example of an unsigned overflow, where the value couldn't be stored in the available no. of bytes. Similarly, for a typical short int, we might get:

$$32760 + 100 = -32676$$

Find out why! Check limits.h for maximum size of short on your computer.

Underflow

Similar behaviour can be observed if the result is less that the minimum (underflow):

$$-32760 - 100 = 32676$$

Rounding

Floating Point Datatypes: Rounding

Since number of bits for representing numbers is limited, real numbers are rounded, e.g. π:

- `float`: $\tilde{\pi} = 3.141592741$,
- `double`: $\tilde{\pi} = 3.141592653589793116$

This might also lead to wrong results:

$$\pi - 3.1415926 \approx 5.36 \cdot 10^{-8}$$

but in single precision one obtains

$$\pi - 3.1415926 = 1.41 \cdot 10^{-7}.$$

This effect is also known as cancellation

Floating Point Datatypes

Limits on finite precision operations

When adding two numbers with a large difference in the exponent, the result might be equal to the larger of the two addend, e.g. in single precision for $x \in \mathrm{R}$:

$$x + 1 \cdot 10^{-8} = x$$

For any floating point type, the smallest number ϵ, such that $1 + \epsilon \neq 1$ is known as the machine precision (check limits.h for your system):

- for `float`: $\epsilon \approx 1.2 \cdot 10^{-7}$,
- for `double`: $\epsilon \approx 2.2 \cdot 10^{-16}$.

Arithmetic and Assignment Operators

Division by Zero and other undefined Operations

With floating point types:

$$x/0.0 = \texttt{INF}$$

for $x \neq 0.0$. INF is a special floating point number for infinity.
For $x = 0.0$:

$$x/0.0 = \texttt{NAN}$$

NAN (not-a-number) is another special floating point number for invalid or not defined results, e.g. square root of negative numbers.
Both operations are (usually) performed without notification, i.e. the program continues execution with these numbers. NANs often occur with uninitialised variables, therefore RAII.
In integer arithmetic, $x/0$ leads to an exception, i.e. the program (usually) aborts.

Von Neumann Architecture: Stored-Program Digital Computer

Data and Instruction both reside in memory (RAM in present context)

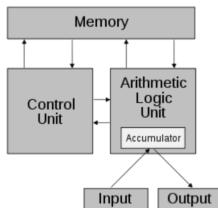

Figure : Schematic of the von Neumann architecture. Ref: http://en.wikipedia.org/wiki/Von_Neumann_architecture

Variables and Datatypes I

In C++, all variables have attributes like: datatype, value, address and scope.

- datatype: char, int, bool, double, float,

  Variable can appear anywhere in the program and declaration syntax is type variablelist;

  - int i,j,k;
  - double x;
  - int *i;
  - bool flag;
  - char c;

  $*i$ declares i to be int pointer which can hold address of a variable.

- Value initialization format:

  - double x =3.0, y(2.0);
  - int m=2, n(3);

Variables and Datatypes II

- Address: Variables have address associated with them which can be accessed by & operator. For example &m will give address of variable m in hexadecimal format.

```
int x = 10;
int y =12;
int *p = & y;
```

Address	Name	Content
2	x	10
6	y	12
16	*p	6

Figure : data representation

Variables and Datatypes III

- Scope: All variables in a C++ program have a domain of visiblity called their scope. For example

```
{
int x = 10;
}
// x is undefined here
```

- can be constants
  - const double pi=3.1415927;
  - const int size = 100;

Good practice to use const as much as possible.

Built-in Data types

Data Types		
Group	Type names	size / precision
Character types	char char16_t char32_t wchar_t	Exactly one byte in size. At least 8 bits. Not smaller than char. At least 16 bits. Not smaller than char16_t. At least 32 bits. Can represent the largest supported character set.
Integer types (signed)	signed char short int long int long long int	Same size as char. At least 8 bits. Not smaller than char. At least 16 bits. Not smaller than short. At least 16 bits. Not smaller than int. At least 32 bits. Not smaller than long int. At least 64 bits.
Integer types (unsigned)	unsigned char unsigned short unsigned unsigned long unsigned long long	Same size as char. At least 8 bits. Not smaller than char. At least 16 bits. Not smaller than short. At least 16 bits. Not smaller than int. At least 32 bits. Not smaller than long int. At least 64 bits.
Floating-point types	float double long double	minimum 4 bytes minimum 8 bytes. Precision not less than float minimum 8 bytes. Precision not less than double
Boolean type	bool	Typically 1 byte
Void type	void	no storage

read about stdint.h, sizeof operator

Integer Constants

- Integral constant: Suffixes specify type
  - int: 0, -3, .
  - unsigned int: 3u, 7U ,$\cdots$
  - short: 0S, -5s,$\cdots$
  - unsigned short: 1us, 9su, 6US,$\cdots$
  - long: 0L, -5l, $\cdots$
  - unsigned long: 1ul, 9Lu, 6Ul, $\cdots$
  - long long: 0LL, -5ll, $\cdots$
  - unsigned long long: 1ull, 9LLu, 6Ull, $\cdots$

Zeroth C++ Program

```
#include<iostream>
int main() {
// to print any message xyz put ''xyz''
std::cout <<"Hello World " <<std::endl;

return 0; }
```

We can compile it using g++ zeroth.C
and run it as ./a.out

Remark

"// denotes begining of a single line Comment."

Zeroth C++ Program

```
#include<iostream>
using namespace std;
int main() {
cout <<"Hello World " << endl;
return 0; }
```

Namespace

What if two variables share name?

C-style solution: Change Name for the second one
Difficult to maintain in large programs

C++ Solution: Namespace

Encapsulates all declarations in a "namespace"!!

```
namespace Box1

{

   int boxSide = 4;
}
```

Sample Code

```
using  namespace std;

namespace Box1{
   int boxSide = 4;
}

namespace Box2{
   int boxSide = 12;
}

int main () {
  cout << Box1::boxSide << endl;
  cout << Box2::boxSide << endl;
 return 0;
}
```

Scope Resolution Operator ::

```
int num = 0;
int main(void) {int num = 0;
::num = 1; // set global num to 1
num = 2; // set local num to 2
return 0; }
```

Expressions and operators

- Arithmetic operators
  - multiplication: a * b
  - division: a / b
  - remainder: a % b
  - addition: a + b
  - subtraction: a - b
  - negation: -a
- Increment and decrement"
  - pre-increment: ++a
  - post-increment: a++
  - pre-decrement: −−a
  - post-decrement: a−−
- Assignment: a = b
- Logical operators(result bool)
  - logical not: !a
  - less than: a < b
  - less than or equal: a <= b
  - greater than: a > b
  - greater than or equal: a >= b
  - equality: a == b
  - inequality: a ! = b
  - logical and: a && b
  - logical or: a || b

First Wrong Code

Can you predict the result of this code?

```
#include<iostream>
main()
{double x, y=10.0;
std::cout<<y/x<<std::endl;
x =0.0;
std::cout<<y/x<<std::endl;
double z =x/y;
std::cout<<z/x<<std::endl;
}
```

Read Goldbergs What Every Computer Scientist Should Know About Floating-Point Arithmetic

C++ Program

```
#include<iostream>
main()
{
int x = 10;
int y =12;
int *p; //p can store any address;
// std::cout<<p; // Code will compile but makes no sense
p = & y; // Now p points to address of y;
std::cout <<x<<", "<<y<<std::endl;
std::cout<<"Address of y is:"<<&y<<std::endl;
std::cout<<"Value of p is:"<<p<<std::endl;
std::cout<<"Address of p is:"<<&p<<std::endl;
}
```

Reference

- Pointer is a very useful tool but requires careful programing!
- Sometime (with functions for example) syntex can easily become ugly!
- Reference is a safer alternate to pointers
- Syntex: starts with & and cannot be left uninitialized!

```
int y = 5;
int &ry; // This is wrong
int &rY = y;
```

- Unlike pointers, once a reference is initialized to an object, it cannot be changed to refer to another object.
- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

Computing Average of numbers

We would Like to compute average of a sequence $\{1, 2, 3\}$

Algo:

- Average =0;
- Average =Average + 1;
- Average = Average +2;
- Average = Average + 3;
- Average = Average/3;

Code: For construct

```
#include<iostream>
main()
{
int data[3] = {1,2,3};
int sum =0;
for(int i=0; i <3; i++)
{sum +=data[i]; }
double average = (double) sum/(double)
3;
std::cout<<"Average =
"<<average<<std::endl;
}
```

For and if Construct I

- For loop has the form

```
for (initialization statement ; checking condition ; increament expression)
statement;
```

- Alternate form is

```
for (initialization statement ; checking condition ; increament expression) {
statement A;
statement B;
...
}
```

- Example:

```
tmp =1; for (int i=1 ; i< 10 ; i++)
tmp *= i; // Equivalent to tmp = tmp *i;
```

- Equivalent but ugly form:

```
for (int i=1, tmp =1 ; i< 10 ; i++, tmp *= i);
```

For and if Construct II

If you like this version may be you should prepare for "The International Obfuscated C Code Contest" !

- Another valid form: useful for infinite loop

  `for ( ; ; ) {··· }`

- Terminating from infinite loop

```
int j, i=0;
for ( ; ; ) {
std::cout<<"iter number is: "<<i<<std::endl;
if(i = 0) //if i is non zero go to next line
std::cout<< " enter a new number:";
else //if i is zero go to next line
std::cout<< " enter a number:";
std::cin>>j;
if((j = 8 && j = 4)) {
std::cout<< " number is not 4 or 8"<<std::endl;
}
else if (j==8) {
```

For and if Construct III

```
std::cout<< " number is 8"<<std::endl;
// jump back to next iteration, so i will not incremented if j == 8
continue;
}

else
// break from loop if j == 4
break;
i++; }
```

Go to Construct I

- Ref: http://blog.julipedia.org/2005/08/using-gotos-in-c.html

```
int foo(···)
{int error;
/* Check function preconditions here */
if (error condition one) {
error = -1;
goto out;
}
if (error condition two) {
error = -2;
goto out;
}
/* Code executed when there are no errors. */
...
error = 0;
out:

return error; }
```

Go to Construct II

- Typically included only for machine produced codes, e.g. FORTRAN $->$ C translators (f2c)
- can always be replaced by one of the other control structures
- Use of any goto in the exercises are not encouraged!

Derived DataType

```
#include<iostream>
struct complex{
double re;
double im; }
or,
class complex{
public: // declaration public ensures data can be accessed like a normal variable
double re;
double im; }
main() {
complex a, b;
std::cout<<"Input Real and Imaginary Part of the complex number"<<std::endl;
std::cin>>a.re>>a.im;
std::cout<<a.re<<", "<<a.im<<" ";
b.re= a.re; b.im= a.im;
std::cout<<std<<endl; std::cout<<b.re<<", "<<b.im<<" ";
}
```

Derived DataType: Public and Private

```
#include<iostream>
class complex {
private://can't be accessed outside class
double amp, theta;
public:
double re, im; };

main() {
complex a;
std::cout<<"Input Real and Imaginary Part of the complex number"<<std::endl;
std::cin>>a.re>>a.im;
std::cout<<a.re<<", "<<a.im<<" ";
std::cout<<std<<endl;
std::cout<<a.amp<<", "<<a.theta<<" "; // This is illegal }
```

Functions

- It is a group of statements that is executed when it is called from some point of the program.
- Similar to variable declaration, you need to declare a function. Syntex is

```
returnType taskName(parameter list);
```

- Return type or parameter can be built in data type(int, double, char etc), or derived data type or pointer/refernce or void (nothing to be returned).
- parameter can be list but only one return type
- declaration just require type for each argument. So a valid for is

```
void doSomething(int, double);
```

- function need to be declared before being used
- definition format is

```
type name ( parameter1, parameter2, ...)  List of statements
```

- Like variable we can get pointer to a function. Example: A function pointer for "doSomething" is

```
void (*fun)(int, double);
```

Computing Average: function

```
#include<iostream>
double average(int data[],int N);
main()
{
int data1[3] = {1,2,3};
int data2[4] = {1,2,3,4};

std::cout<<"Average of 1= "<<average(data1,3)<<std::endl;
std::cout<<"Average of 2= "<<average(data2,4)<<std::endl;
}
double average(int data[],int N){int sum =0;
for(int i=0; i <N; i++)
{sum +=data[i]; }
double answer = (double) sum/(double)N; return answer; }
```

Function Overloading in C++

```
#include<iostream>
double average(int data[],int N); double average(double data[],int N);
main()
{
int data1[3] = {1,2,3};
double data2[4] = {1.,2.,3.,4.};

std::cout<<"Average of 1= "<<average(data1,3)<<std::endl;
std::cout<<"Average of 2= "<<average(data2,4)<<std::endl;
}
double average(int data[],int N){int sum =0;
for(int i=0; i <N; i++)
{sum +=data[i]; }
double answer = (double) sum/(double)N; return answer; }
double average(double data[],int N){double sum =0;
for(int i=0; i <N; i++)
{sum +=data[i]; }
double answer = sum/(double)N; return answer; }
```

Derived DataType: Constructor

```
#include<iostream>
struct complex{


// constructor has same name as class with no return type
complex(double a=0.0, double b=0.0) // Default value for complex number is 0.0
{re=a ; im = b; };
double re;
double im; }


main() {
complex a(3,4), b;
std::cout<<"Input Real and Imaginary Part of the complex number"<<std::endl;
std::cin>>a.re>>a.im;
std::cout<<a.re<<", "<<a.im<<" ";
std::cout<<std<<endl; std::cout<<b.re<<", "<<b.im<<" ";
}
```

Organizing code

- Reusability: A good library can be quite useful!
- Concept of header file: Put useful derived data type declaration in header file and its implementation in C file.
- Make sure that same code is not included twice: conditional compilation Model

```
#ifndef NAME
#define NAME
// All code goes here
#endif
```

- put your header file in a directory X and compile with -I X

Stage of Compilations

Try g++ -save-temps zeroth.C

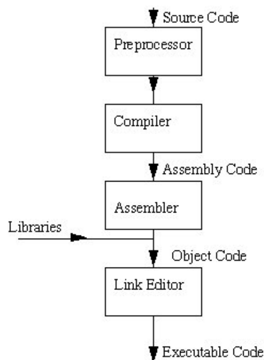

Figure : Compilation Stages

Average of int and double: c-style typedef Solution

```
#include<iostream> //create an alias for double
typedef double dataType;
double average(dataType data[],int N);
main()
{
dataType data2[4] = {1.51,2.51,3.51,4.51};

std::cout<<"Average of 2= "<<average(data2,4)<<std::endl;
}


double average(dataType data[],int N){T sum =0;
for(int i=0; i <N; i++)
{sum +=data[i]; }
double answer = (double) sum/(double)N; return answer; }
```

Average of int and double: Templates Solution

```
#include<iostream>
template<typename T>
double average(T data[],int N);
main()
{
int data1[3] = {1,2,3};
double data2[4] = {1.51,2.51,3.51,4.51};

std::cout<<"Average of 1= "<<average(data1,3)<<std::endl;
std::cout<<"Average of 2= "<<average(data2,4)<<std::endl;
}

template<typename T> double average(T data[],int N){T sum =0;
for(int i=0; i <N; i++)
{sum +=data[i]; }
double answer = (double) sum/(double)N; return answer; }
```

Operator Overloading

- Class or struct are like sets, which can have distinct elements
- However sets in math are often endowed with operations. For example for two complex number $z_1 = a_1 + Ib_1$ and $z_2 = a_2 + Ib_2$, $+$ is defined as $z_1 + z_2 = (a_1 + a_2) + I(b_1 + b_2)$
- C++ allow us to define certain operators to be redefined in context of a given class. This property will allow us to write code like this:

```
Complex a(1.8,2.4);
Complex b(2.1,3.2);
Complex c = a+b;
```

A simple example to do such overloading is

```
struct complex{
complex(double a=0.0, double b=0.0)
{re=a ; im = b; };
double re;
double im; }

const complex operator+(complex &first,
complex &second) {
complex result;
result.re = first.re + second.re;
result.im = first.im + second.im;
return result; }
```

Assignment Operator

- Often copying one object with other implies member wise copy. For example:

```
Complex a(1.8,2.4), b(0.1,0.2);
b =a;
```

should imply

$$b = (1.8, 2.4).$$

- $=$ operator can be overloaded as

```
#include<iostream>
struct complex{
...
complex& operator= (const complex
&rhs){
re= rhs.re; im=rhs.im;
return *this; //Current Object }
}
```

- A simple example using complex number class would be now

```
main(){Complex a(1.8,2.4),
b(0.1,0.2); complex c; c =a+b;
}
```

- Notice that compiler generate a assignment operator by default. So explicit overloading in this particular example was not needed.
- Role of default assignment operator is not correct in this example

```
#include<iostream>
struct A{
... double *b; }
```

Copy Constructor

- We might to create new object from existing object

```
Complex a(1.8,2.4), b(0.1,0.2);
complex c(a); complex d = c;
```

  which should again imply element wise copy.

- copy constructor can be written as

```
#include<iostream>
struct complex{
...
complex (const complex &rhs){
re= rhs.re; im=rhs.im;
}
}
```

- Notice that compiler generate a copy constructor by default. So explicit creation in this particular example was not needed.
- Role of default copy constructor is not correct in this example

```
#include<iostream>
struct A{
... double *b; }
```

Abstraction

```
#include<iostream>
struct complex{
...
// We will discuss operator & later
complex& operator= (const complex &rhs){
re= rhs.re; im=rhs.im;
return *this; //Current Object }
}; // End of dataType def

//const because printing is not suppose
to change the numbers
std::ostream& operator<< (std::ostream&
os, const complex &num){
os<<num.re<<", "<<num.im<<"";
return os;
}
```

```
std::istream& operator>> (std::istream&
os, complex &num){
os>>num.re>>num.im;
return os;
}
main() {
complex a(3,4), b;
std::cout<<"Input complex number:
"<<std::endl;
std::cin<<a;
std::cout<<"Number is: "<<a<<std::endl ;

b=a;
std::cout<<"Number is: "<<b<<std::endl;

}
```

More Abstraction

```
const complex operator+(complex &first,
complex &second) {
complex result;
result.re = first.re + second.re;
result.im = first.im + second.im;
}

const complex operator*(const complex
&first,const complex &second) {
complex result;
result.re = first.re * second.re-
first.im *second.im;
result.im = first.re * second.im +
first.im *second.re;
return result;
}
```

```
main() {
complex a(3,4), b(2,3);
std::cout<<"First Number is
:"<<a<<std::endl;
std::cout<<"Second Number is
:"<<b<<std::endl;
complex c=a+ b;
std::cout<<"Sum is :"<<c<<std::endl;
complex d=a* b;
std::cout<<"Multiplication is
:"<<d<<std::endl;
}
```

Creating your own Library for Complex Number

- Open a file called "complex.h"

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_
// All code goes here
#endif
```

- Often used *.o files can be packed into a library, e.g.:

```
ar ruc libtest.a complex.o
ranlib libtest.a
g++ stub.C -L. -ltest
```

- ar creates an archive, more than one object file can be specified
- The name must be libsomething.a
- ranlib adds a table of contents (not needed on some platforms)
- -L specifes the directory where the library
- -lsomething specifies looking in the librarylibsomething.a

Array Class I

```
struct vectInt{
vectInt(int size)
{
N=size ;
std::cout<<"Enter "<<N<<" numbers: ";
std::cin>>(*data);
}
int operator[](const int i) const {return data[i];}
int & operator[](const int i) {return data[i];}

double average(){
int sum =0;
for(int i=0; i <N; i++) sum +=data[i];
double answer = (double) sum/(double)N;
return answer;
}
```

Array Class II

```
int N;

int data[ ];
}// end of struct

std::istream& operator>> (std::istream& os,const vectInt &myVect)
{for(int i=0; i<myVect.N; i +=1)
os>>myVect.data[i];

return os; }

main() {
vectInt data1(3);
std::cout>>data1;
std::cout<<"Average of 1= "<<average(data1,3)<<std::endl; }
```

Array Class revisted I

```
#include<iostream>
template<typename T, int N>
struct vectType
{
vectType()
{
std::cin>>(*data);
}
// operator [] is overloaded in pairs typically

T operator[](const int i) const {return data[i];}
T & operator[](const int i) {return data[i];}

friend std::istream& operator>> (std::istream& os, vectType &thisVec);
double average(){
int sum =0;
for(int i=0; i <N; i++)
```

Array Class revisted II

```
sum +=data[i];
double answer = (double) sum/(double)N;
return answer;
}

// Now data inside derived dataType

static int size=N;

int data[N];

}// End of derived data type declaration
std::istream&
operator>> (std::istream& os,const vectType &myVect)
{
for(int i=0; i<myVect.N; i +=1)
os>>myVect.data[i];
```

Array Class revisted III

```
return os; }

int main()
{
vectType<double 3> data1();
std::cout>>data1;
std::cout<<"Average of 1= "<<average(data1,3)<<std::endl;
return 0;
}
```

What about vectors of Complex numbers: Trait Type

- Result of averaging is not always double
- Except int for any other type T average return type T
- Trait as return type for average

```
// the general traits type:
template <class T>
struct averageType {typedef T type; };
// the special cases:
template< >
struct averageType <int> {
typedef double type;};
```

- Now better definition of average function is

```
template<typename T>
// :: Allows to access things belonging to struct
typename averageType<T>::type
average(T data[],int N){
averageType<T>::type sum =0;
for(int i=0; i <N; i++) {sum +=data[i]; }
return sum/N; }
```

Callback Inlining Techniques

```
double integrate (double a, double b, int numSamplePoints,double (*f) (double))
{
double delta = (b -a)/(numSamplePoints-1.0);
double sum = 0.0;
for(int i =0 ; i<numSamplePoints ;i++)
sum += f(a+ i*delta)
return sum*(b-a)/numSamplePoints;
}
```

This solution works but inefficient!

Callback Inlining Techniques: Template Based Solution

```
struct myFun{
double operator(double x){
return sin(x);
}
}
template<typename T>
double integrate (double a, double b, int numSamplePoints, T f)
{
double delta = (b -a)/(numSamplePoints-1.0);
double sum = 0.0;
for(int i =0 ; i<numSamplePoints ;i++)
sum += f(a+ i*delta)
return sum*(b-a)/numSamplePoints;
}
```

Factorial via recursive functions

```
int factorial( int n)
{if (n == 0) return 1;
return n * factorial(n - 1);
}
int main() {
std::cout<<factorial(15)<<std::endl;
return 0; }
```

Often difficult to optimize recursive functions

Compile Time Computations

```
template <int N>
struct Factorial {
// Enum is like int but does not require memory
enum {value = N * Factorial<N - 1>::value }; };

// Take care of factorial 0
template <>
struct Factorial<0>
{enum  value = 1 ; };


int main() {// Ans is computed even before code runs
int x = Factorial<4>::value;
}
```

Loop Unrolling

- The unrolled loop is optimal for smaller vectors

```
inline double dot3(const double* a, const double* b) {
return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

- Question: how can we unroll for vectors of size N ?
- Answer: template meta programming

Meta program for loop unrolling I

```
// The dot() function invokes metaDot
template<class T, int N>
inline T dot(TinyVector<T,N>& a, TinyVector<T,N>& b)
{
return metaDot<N-1>::f(a,b);
}


template<int m>
struct metaDot {
template<class T, int N>
static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
{
return a[m]*b[m] + metaDot<m-1>::f(a,b);
}
};
```

Meta program for loop unrolling II

```
template<> // the end of the recursion
struct metaDot<0> {

template<class T, int N>
static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
{
return a[0]*b[0];
}
};


int main() {
TinyVector<double,4> a, b;
double r = dot(a,b);
}
```

Actual code created by compiler can be understood as

Meta program for loop unrolling III

```
double r = dot(a,b);
= metaDot<3>::f(a,b);
= a[3]*b[3] + metaDot<2>::f(a,b);
= a[3]*b[3] + a[2]*b[2] + metaDot<1>::f(a,b);
= a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + metaDot<0>::f(a,b);
= a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + a[0]*b[0];
```

Some Useful Jargons

- Binary data: Data or instructions translated into sequences of 0 and 1.
- Bit: Quantity that can have only two possible values
- Bus Width: The size (in bits) of the packet of data which is processed (e.g. moved) in each work cycle.
- Cache: A temporary storage or buffer
- Gigahertz (GHz): 10^9 cycles per second
- Clock frequency: The frequency (usually in GHz) at which a CPU is running
- Cycles per instruction (cip): the number of clock cycles that happen when an instruction is being executed. Ex: Classic RISC processor with
  - Instruction fetch cycle (IF)
  - Instruction decode/Register fetch cycle (ID)
  - Execution/Effective address cycle (EX)
  - Memory access (MEM)
  - Write-back cycle (WB)