3. Object-oriented programming

(part. 2)

Game Player Experience Design

Prof. H. Kang

C++

C++ is known to be a very powerful language.

- C++'s execution speed and ability to cheaply use resources surpass other languages.
- Thanks to C++'s performance, it is widely used to develop game engines, games, and desktop applications.

One of C++'s advantages is how scalable it could be.

- C++ allows you to control a lot of how you manage system resources.
- You can directly manage the memory, thread, storages, etc.

C++

Therefore, many apps that are very resource intensive are usually built with C++.

C++ has a huge community and very big job market.

- Community size is important, because the larger a programming language community is, the more support you would be likely to get.
- C++ has a very big job market as it is used in various industries like game, finance, server management, metaverse, design tool, etc.

C++

This lecture assumes that students already know the basics of C++.

- Therefore, many basic concepts are skipped to be lectured.

Instead, we focus on the several ‘C++ extensions’ that are helpful to understand engine code.

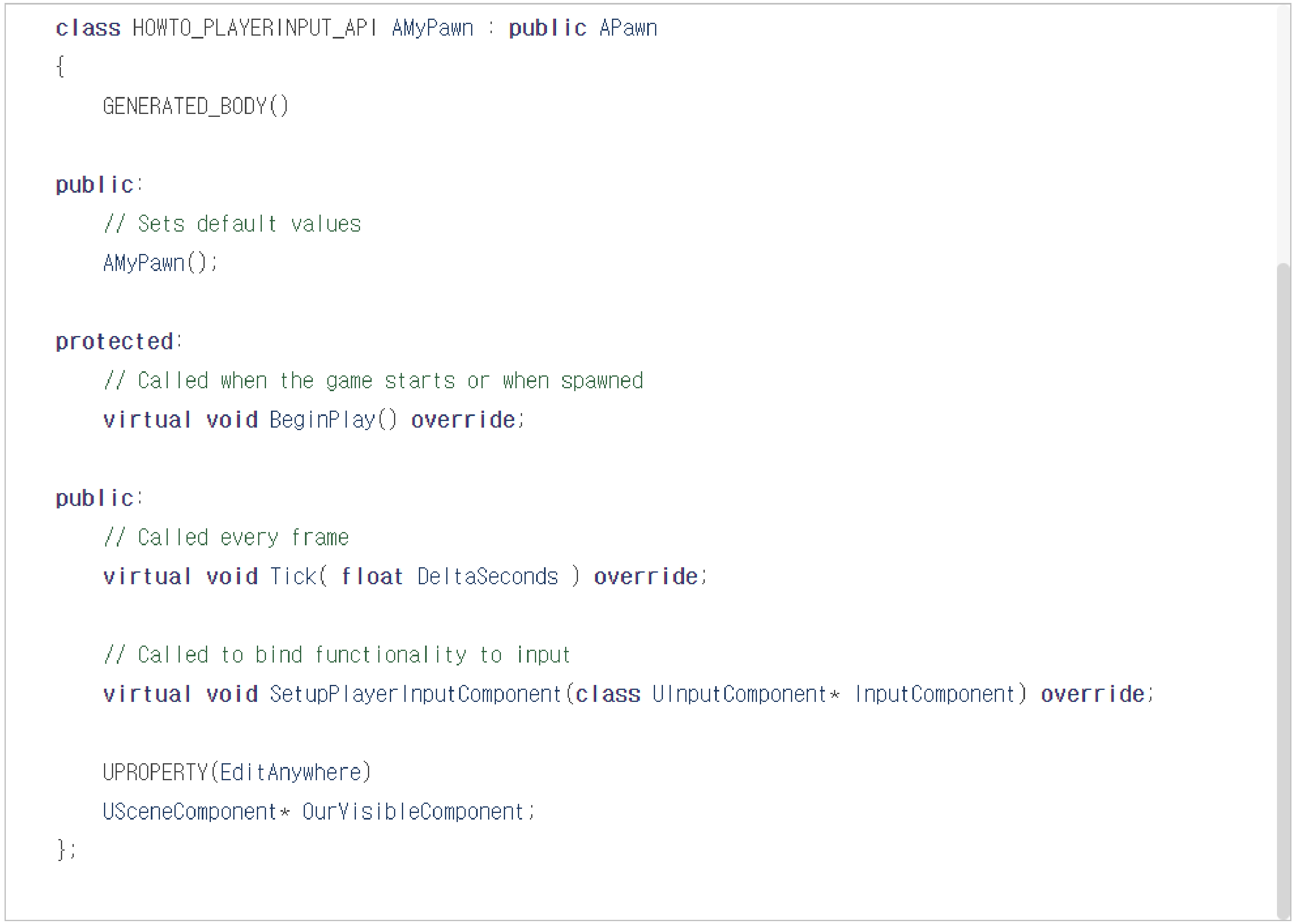
- Of course, we cannot handle all extensions.
- If you are interested in the C++ language and want to learn more, googling will be helpful.

Engine code?

- Game engine includes a lot of code chunks.
- Literally a lot lot lot of.
- To understand its structure and functionalities, you have to know extensions of C++ language and widely used coding pattern.

C++

MyPawn.h



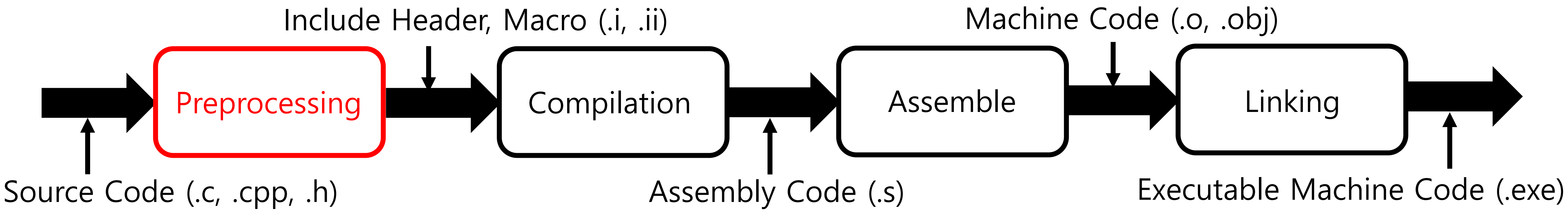
```cpp
class HOWTO_PLAYERINPUT_API AMyPawn : public APawn
{
    GENERATED_BODY()

public:
    // Sets default values
    AMyPawn();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent* InputComponent) override;

    UPROPERTY(EditAnywhere)
    USceneComponent* OurVisibleComponent;
};
```

Unreal Engine Code Sample - MyPawn.h

C++ preprocessor

The preprocessor performs preliminary operations on C++ code before they are passed to the compiler.

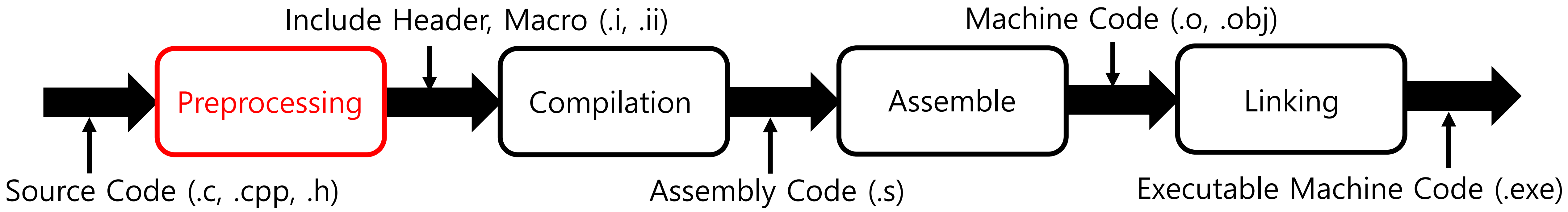
- The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.


Include Header, Macro (.i, .ii)
Machine Code (.o, .obj)
Preprocessing
Compilation
Assemble
Linking
Source Code (.c, .cpp, .h)
Assembly Code (.s)
Executable Machine Code (.exe)

C++ preprocessor

The preprocessing begins with the following four phases:

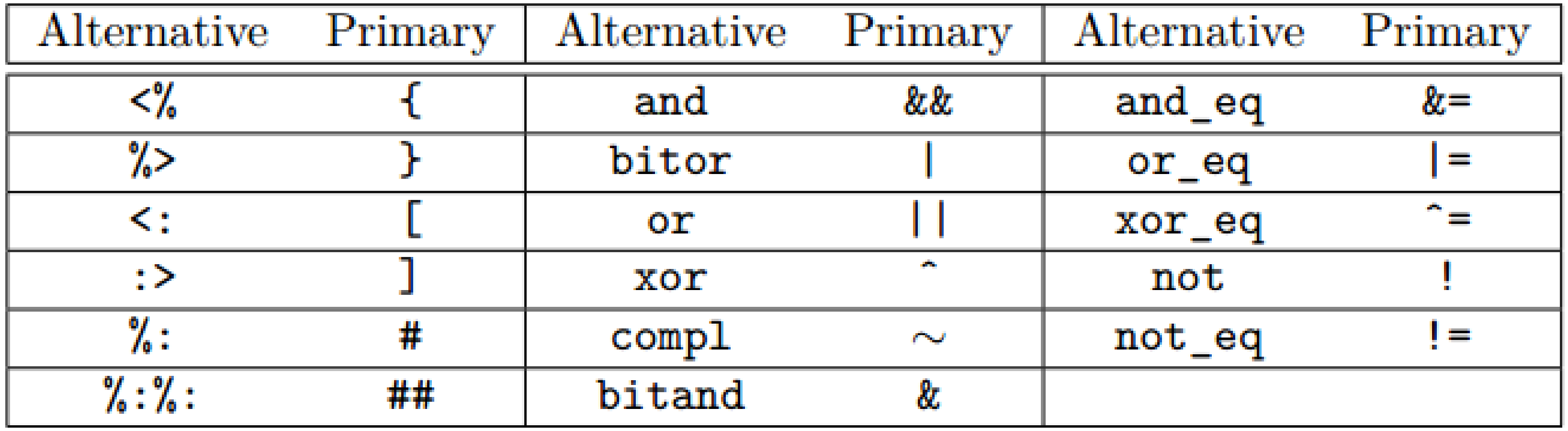
- **Trigraph replacement**: The preprocessor replaces trigraph sequences with the characters they represent. (e.g. ??(= [, ??! = |)
- **Line splicing**: Physical source lines that are continued with escaped newline sequences are spliced to form logical lines.
- **Tokenization**: The preprocessor breaks the result into preprocessing tokens and whitespace. It replaces comments with whitespace.
- **Macro expansion and directive handling**: Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros and, since the 1999 version of the C standard, handles _Pragma operators.


Include Header, Macro (.i, .ii)
Machine Code (.o, .obj)
Preprocessing
Compilation
Assemble
Linking
Source Code (.c, .cpp, .h)
Assembly Code (.s)
Executable Machine Code (.exe)

C++ preprocessor

The preprocessing begins with the following four phases:

- Trigraph replacement example:

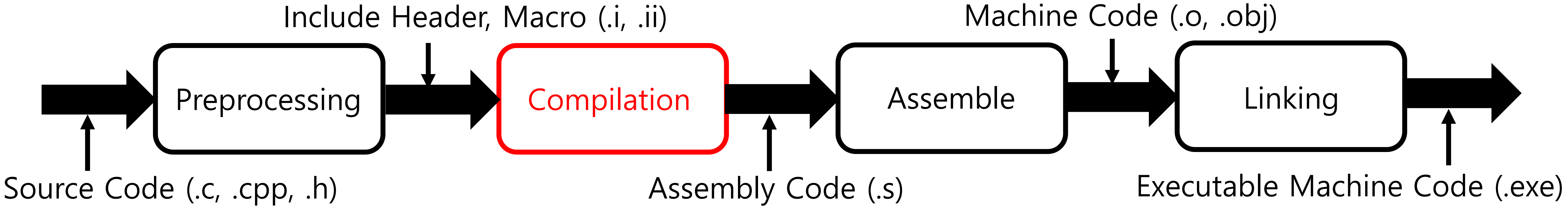


Alternative	Primary	Alternative	Primary	Alternative	Primary
<%	{	and	&&	and_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

- Pragma example:
  - #pragma once - the current source file to be included only once in a single compilation.
  - #pragma comment(linker, "/include:__mySymbol") - Places a linker option in the object file.

C++ compiler

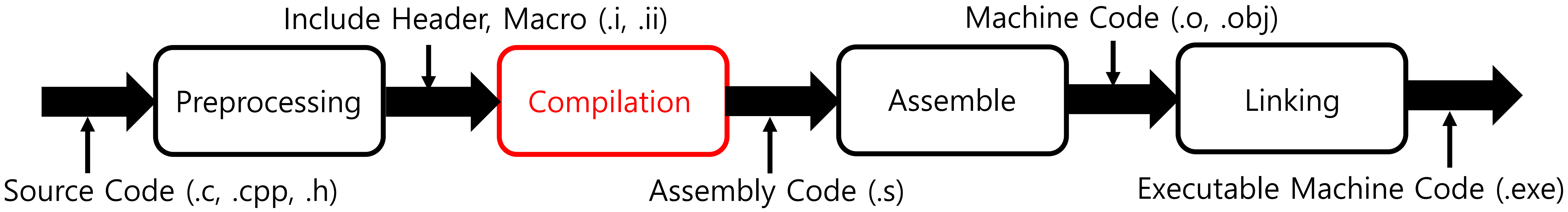
A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.


Include Header, Macro (.i, .ii)
Machine Code (.o, .obj)
Preprocessing
Compilation
Assemble
Linking
Source Code (.c, .cpp, .h)
Assembly Code (.s)
Executable Machine Code (.exe)

C++ compiler

The time duration in which the programming code is converted to the machine code is called compile time (or compile-time).

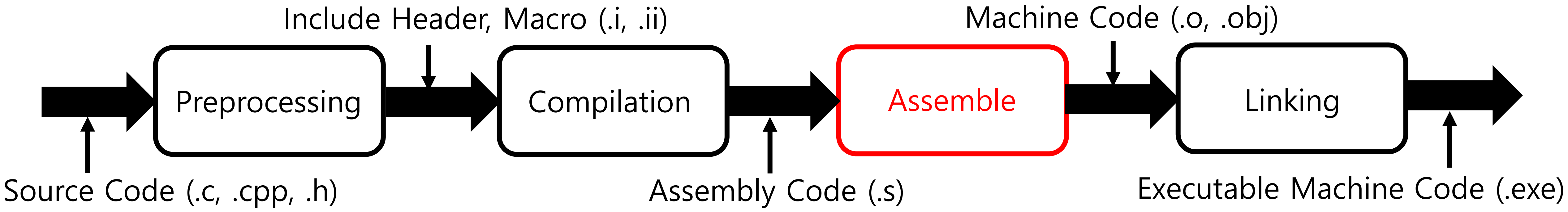
- The compile time usually include semantic analysis and code generation.
- During the compile time, compiler detects compile-time errors: syntax errors and semantic errors.
  - Syntax errors: when the programmer does not follow the syntax of programming language, then the compiler will throw the syntax error. (e.g. missing semicolon)
  - Semantic errors: when the statements are not meaningful to the compiler, then the semantic error exists.
    (e.g. a + b = c)
- When compile-time errors are detected, the compiler prevents the code from execution.


Include Header, Macro (.i, .ii)
Machine Code (.o, .obj)
Preprocessing
Compilation
Assemble
Linking
Source Code (.c, .cpp, .h)
Assembly Code (.s)
Executable Machine Code (.exe)

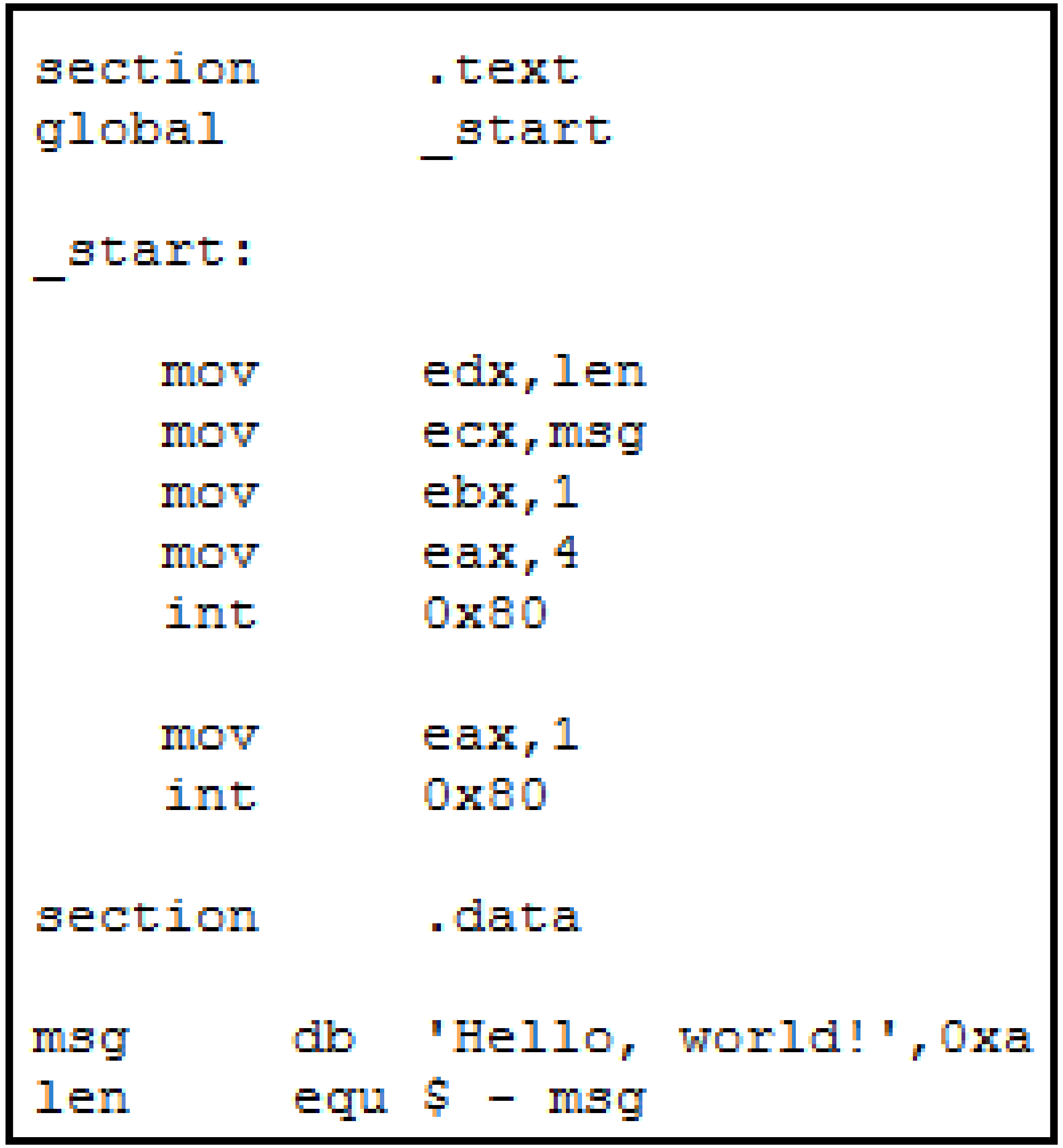
C++ assembler

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.

- The computational step when an assembler is processing a program is called assembly time.
- Assembler creates object code (.o, .obj) by translating assembly language.

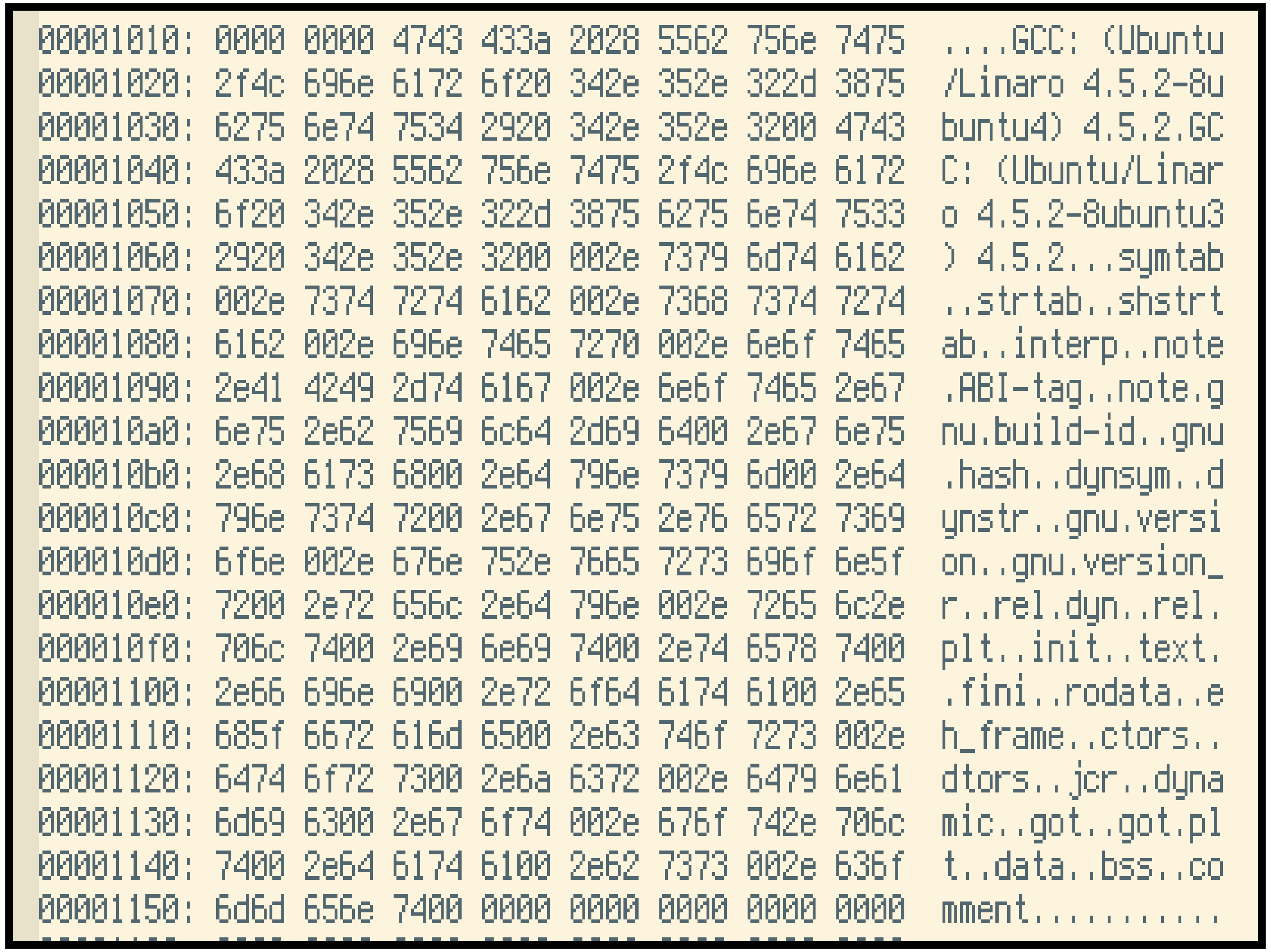

Include Header, Macro (.i, .ii)
Machine Code (.o, .obj)
Preprocessing
Compilation
Assemble
Linking
Source Code (.c, .cpp, .h)
Assembly Code (.s)
Executable Machine Code (.exe)

C++ assembler



```
section     .text
global      _start

_start:

    mov     edx,len
    mov     ecx,msg
    mov     ebx,1
    mov     eax,4
    int     0x80

    mov     eax,1
    int     0x80

section     .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Assembly code sample



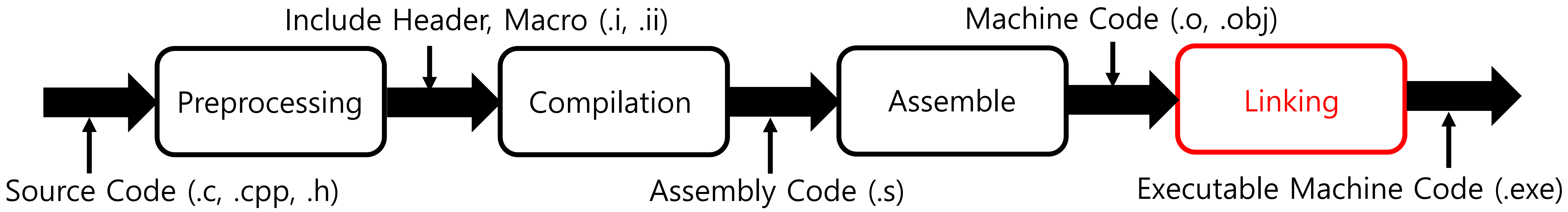
```
00001010: 0000 0000 4743 433a 2028 5562 756e 7475  ....GCC: (Ubuntu
00001020: 2f4c 696e 6172 6f20 342e 352e 322d 3875  /Linaro 4.5.2-8u
00001030: 6275 6e74 7534 2920 342e 352e 3200 4743  buntu4) 4.5.2.GC
00001040: 433a 2028 5562 756e 7475 2f4c 696e 6172  C: (Ubuntu/Linar
00001050: 6f20 342e 352e 322d 3875 6275 6e74 7533  o 4.5.2-8ubuntu3
00001060: 2920 342e 352e 3200 002e 7379 6d74 6162  ) 4.5.2...symtab
00001070: 002e 7374 7274 6162 002e 7368 7374 7274  ..strtab..shstrt
00001080: 6162 002e 696e 7465 7270 002e 6e6f 7465  ab..interp..note
00001090: 2e41 4249 2d74 6167 002e 6e6f 7465 2e67  .ABI-tag..note.g
000010a0: 6e75 2e62 7569 6c64 2d69 6400 2e67 6e75  nu.build-id..gnu
000010b0: 2e68 6173 6800 2e64 796e 7379 6d00 2e64  .hash..dynsym..d
000010c0: 796e 7374 7200 2e67 6e75 2e76 6572 7369  ynstr..gnu.versi
000010d0: 6f6e 002e 676e 752e 7665 7273 696f 6e5f  on..gnu.version_
000010e0: 7200 2e72 656c 2e64 796e 002e 7265 6c2e  r..rel.dyn..rel.
000010f0: 706c 7400 2e69 6e69 7400 2e74 6578 7400  plt..init..text.
00001100: 2e66 696e 6900 2e72 6f64 6174 6100 2e65  .fini..rodata..e
00001110: 685f 6672 616d 6500 2e63 746f 7273 002e  h_frame..ctors..
00001120: 6474 6f72 7300 2e6a 6372 002e 6479 6e61  dtors..jcr..dyna
00001130: 6d69 6300 2e67 6f74 002e 676f 742e 706c  mic..got..got.pl
00001140: 7400 2e64 6174 6100 2e62 7373 002e 636f  t..data..bss..co
00001150: 6d6d 656e 7400 0000 0000 0000 0000 0000  mment...........
```

Object code sample

C++ linker

A linker or link editor is a computer system program that takes one or more object files and combines them into a single executable file, library file, or another "object" file.

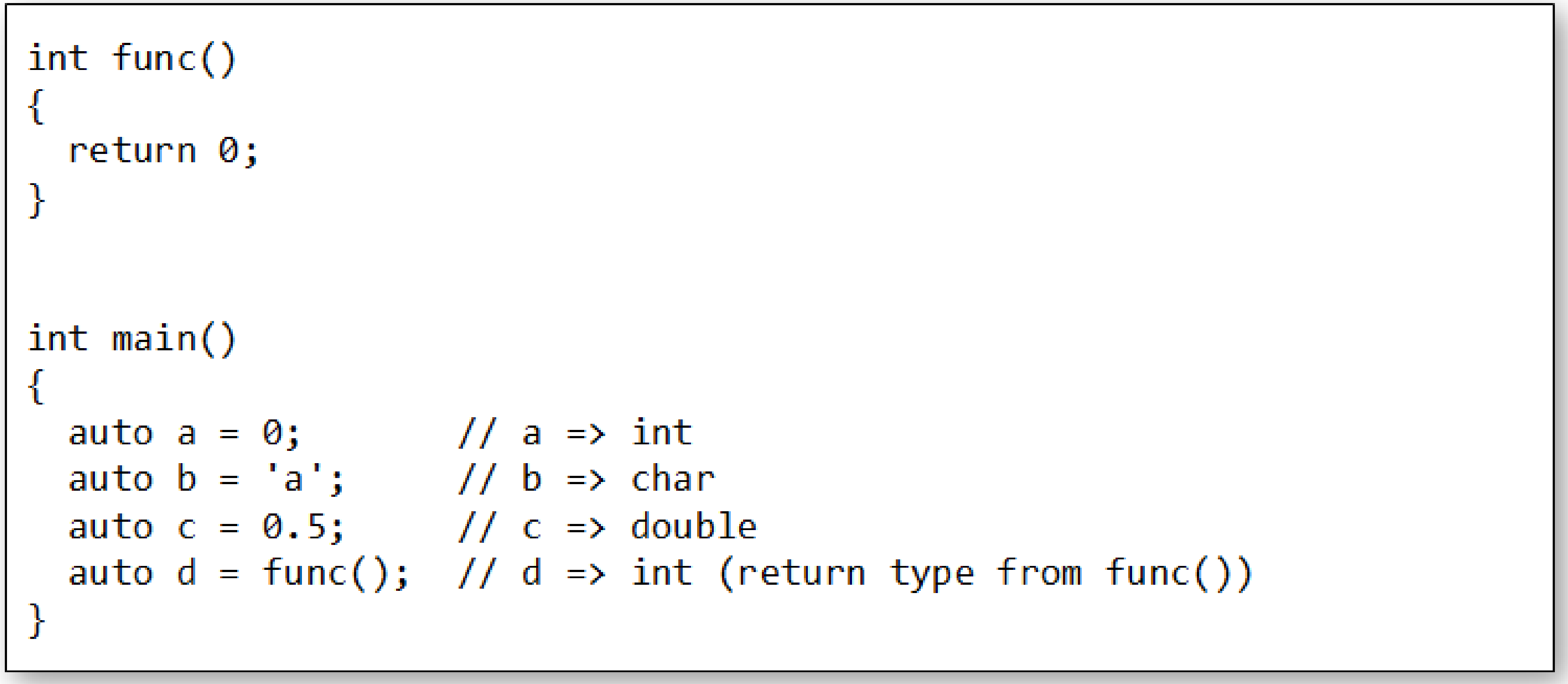
- Computer Programs typically are composed of several parts or modules and these parts/modules do not need to be contained within a single object file.
- The job of linker is three fold:
  - The linker combines all the object files into a single executable program.
  - The linker also link library file that is a collection of precompiled code.
  - The linker makes sure all cross-file dependencies are resolved properly. (For example, dependencies between multiple .cpp files)


Include Header, Macro (.i, .ii)
Machine Code (.o, .obj)
Preprocessing
Compilation
Assemble
Linking
Source Code (.c, .cpp, .h)
Assembly Code (.s)
Executable Machine Code (.exe)

Auto

Auto

- The auto keyword directs the compiler to use the initialization expression of a declared variable, or lambda expression parameter, to deduce its type.



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


int main()
{
  auto a = 0;       // a => int
  auto b = 'a';     // b => char
  auto c = 0.5;     // c => double
  auto d = func();  // d => int (return type from func())
}
```

auto example[auto]

Auto

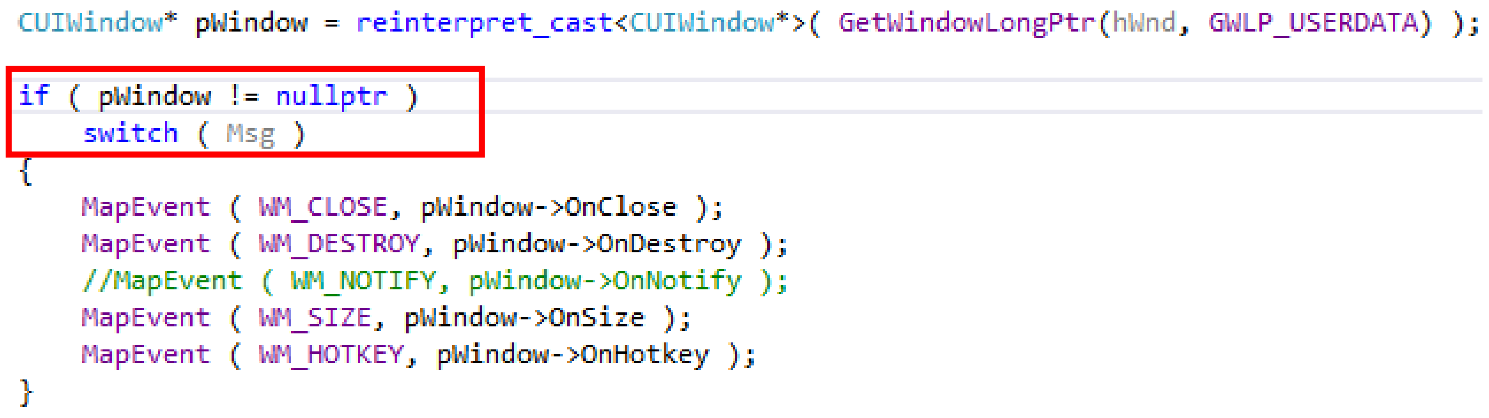
Auto

- Auto provides following benefits:
  - Robustness: If the expression’s type is changed - this includes when a function return type is changed - it just works.
  - Performance: You’re guaranteed that there will be no conversion.
  - Usability: You don't have to worry about type name spelling difficulties and typos.
  - Efficiency: Your coding can be more efficient.
- However, abuse of auto can increase the potential for code to behave in an unintended ways.

C++

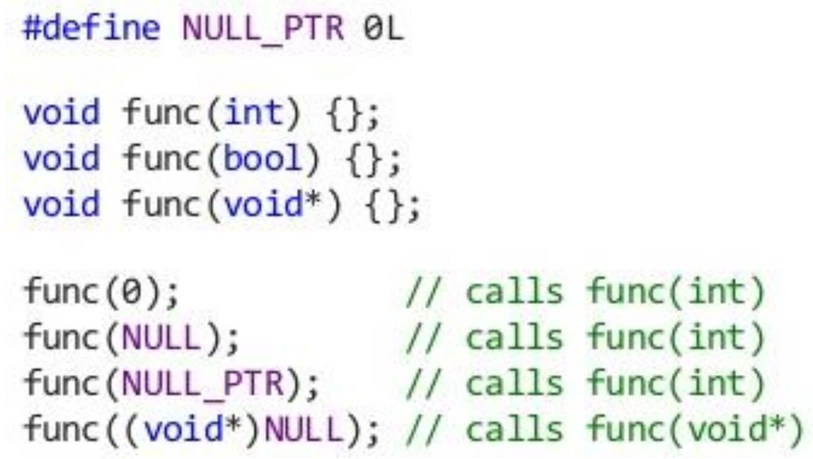
Nullptr

- In early versions of C and C++, a Null pointer was specified by using the literal 0, sometimes cast to (void*) or (char*).
- This lacked type safety and could cause problem because of C/C++'s implicit integer conversions.
- Since C++11, the type-safe explicit literal value *nullptr* is used to represent a null pointer.



```
CUIWindow* pWindow = reinterpret_cast<CUIWindow*>( GetWindowLongPtr(hWnd, GWLP_USERDATA) );

if ( pWindow != nullptr )
    switch ( Msg )
{
    MapEvent ( WM_CLOSE, pWindow->OnClose );
    MapEvent ( WM_DESTROY, pWindow->OnDestroy );
    //MapEvent ( WM_NOTIFY, pWindow->OnNotify );
    MapEvent ( WM_SIZE, pWindow->OnSize );
    MapEvent ( WM_HOTKEY, pWindow->OnHotkey );
}
```

nullptr[null]



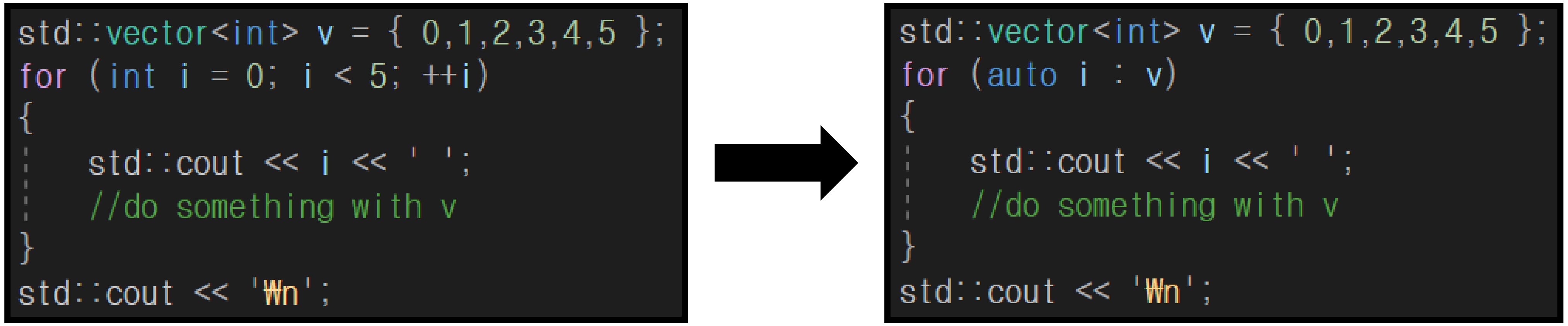
```
#define NULL_PTR 0L

void func(int) {};
void func(bool) {};
void func(void*) {};

func(0);            // calls func(int)
func(NULL);         // calls func(int)
func(NULL_PTR);     // calls func(int)
func((void*)NULL);  // calls func(void*)
```

overloading resolution rule

C++

Range-based for Loops

- Since C++11, for statement have extended to support a short-hand “foreach” declaration style.
- This allows you to iterate over C-style arrays and any other data structure for which the non-member begin() and end() functions are defined.



```
std::vector<int> v = { 0,1,2,3,4,5 };
for (int i = 0; i < 5; ++i)
{
    std::cout << i << ' ';
    //do something with v
}
std::cout << '₩n';
```

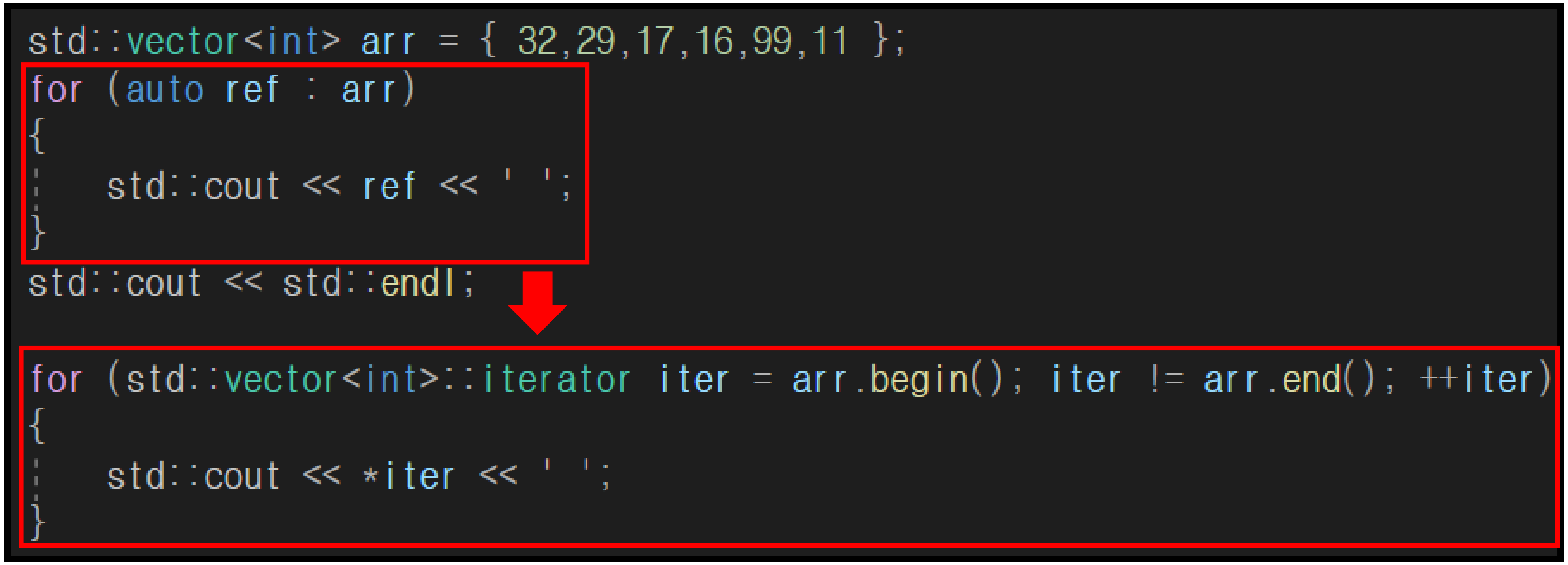
```
std::vector<int> v = { 0,1,2,3,4,5 };
for (auto i : v)
{
    std::cout << i << ' ';
    //do something with v
}
std::cout << '₩n';
```

range-based for loop[for]

C++

Iterator

- Iterators are used to point at the memory addresses of STL containers.
- They reduce the complexity and execution time of program.



```
std::vector<int> arr = { 32,29,17,16,99,11 };
for (auto ref : arr)
{
    std::cout << ref << ' ';
}
std::cout << std::endl;

for (std::vector<int>::iterator iter = arr.begin(); iter != arr.end(); ++iter)
{
    std::cout << *iter << ' ';
}
```

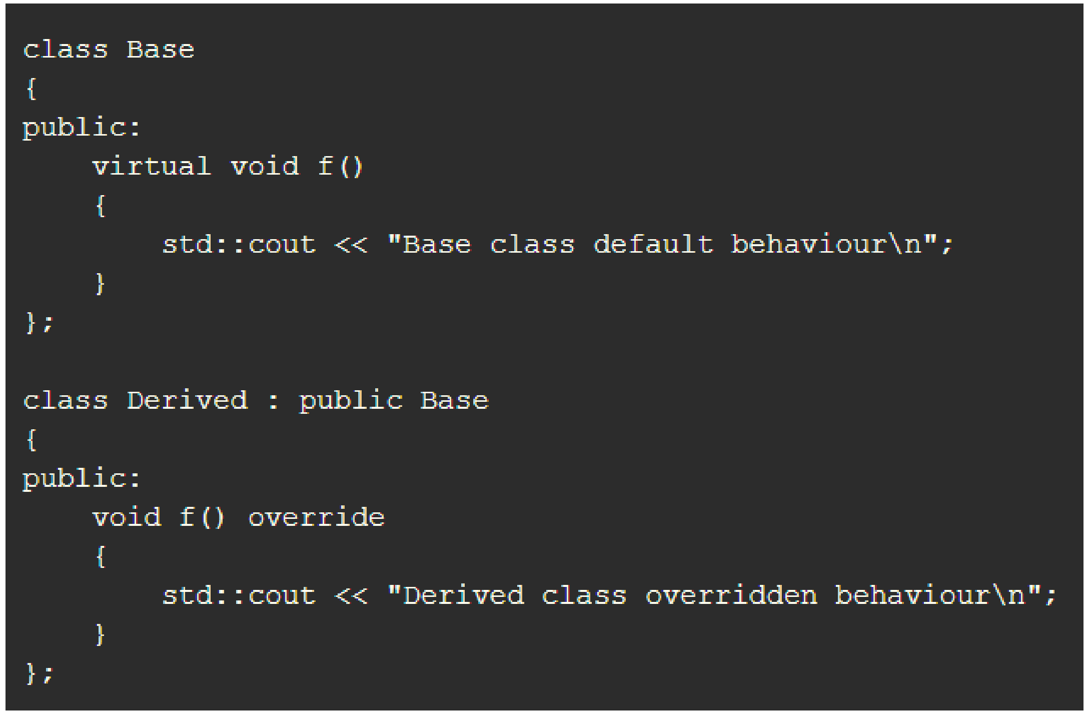
range-based for loop[iter]

C++

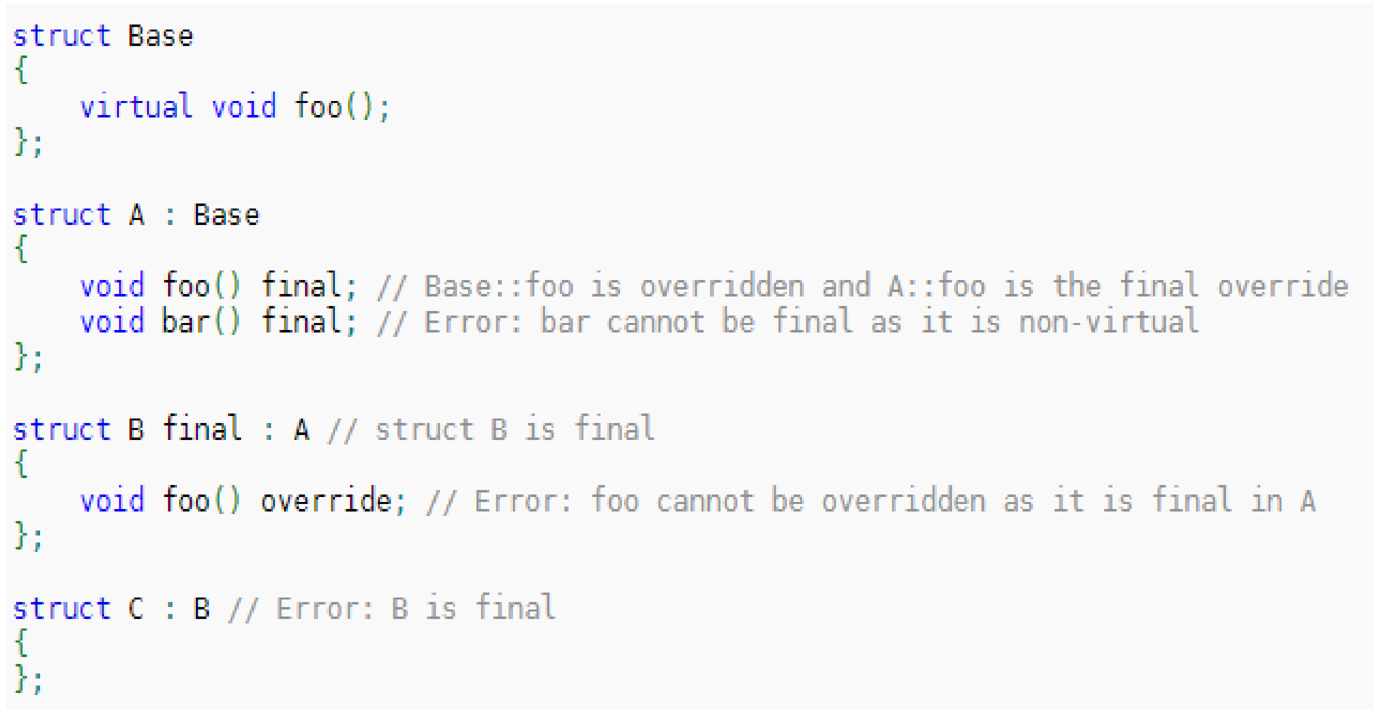
Virtual, final and override

- C++11 added two keywords that allow to better express your intentions with what programmers want to do with virtual functions: *override* and *final*.
- They allow to express programmers’ intentions both to code readers as well as to the compiler.
- *Override* is a feature to use without moderation. Every time you define a method in the derived class that overrides a virtual method in the base class, you should tag it *override*.
- *Final* specifies that a virtual function cannot be overridden in a derived class.

Virtual, final and override



```
class Base
{
public:
    virtual void f()
    {
        std::cout << "Base class default behaviour\n";
    }
};

class Derived : public Base
{
public:
    void f() override
    {
        std::cout << "Derived class overridden behaviour\n";
    }
};
```



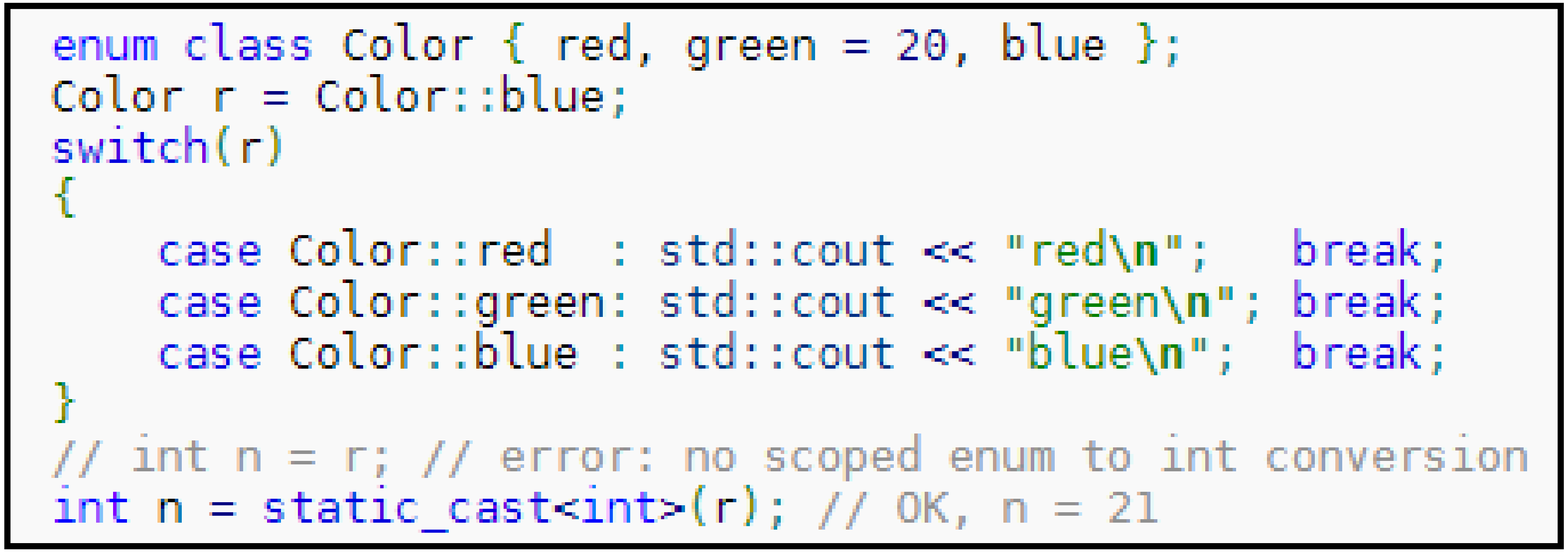
```
struct Base
{
    virtual void foo();
};

struct A : Base
{
    void foo() final; // Base::foo is overridden and A::foo is the final override
    void bar() final; // Error: bar cannot be final as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B // Error: B is final
{
};
```

override & final[over][final]

C++

Enum

- Enumerators, declared using the keywords *enum* class, scopes its enumerators just like a class or struct scopes its members, and permits the programmer to specify the underlying type.
- enum example



```cpp
enum class Color { red, green = 20, blue };
Color r = Color::blue;
switch(r)
{
    case Color::red  : std::cout << "red\n";   break;
    case Color::green: std::cout << "green\n"; break;
    case Color::blue : std::cout << "blue\n";  break;
}
// int n = r; // error: no scoped enum to int conversion
int n = static_cast<int>(r); // OK, n = 21
```

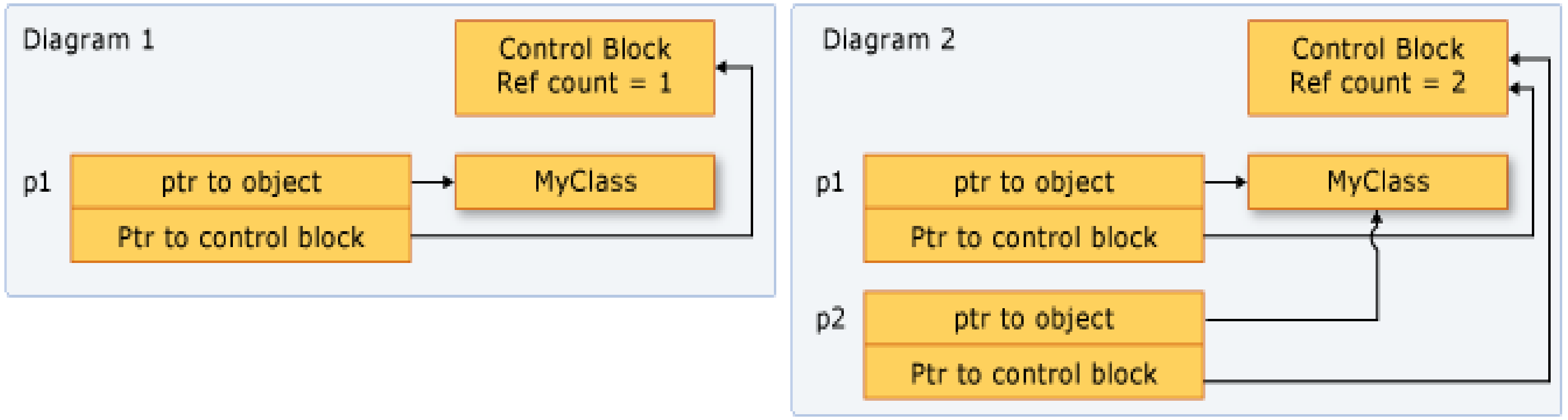
Standardized Smart Pointers

- A smart pointer is an object that acts like a pointer, but additionally provides control on construction, destruction, copying, moving and dereferencing.
- They have a number of advantages over regular pointers. Indirection through a null pointer is checked. No delete is ever necessary. Objects are automatically freed when the last pointer to them has gone away.
- One significant problem with these smart pointers is that unlike regular pointers, they don't respect inheritance. Smart pointers are unattractive for polymorphic code.

C++

Standardized Smart Pointers Examples

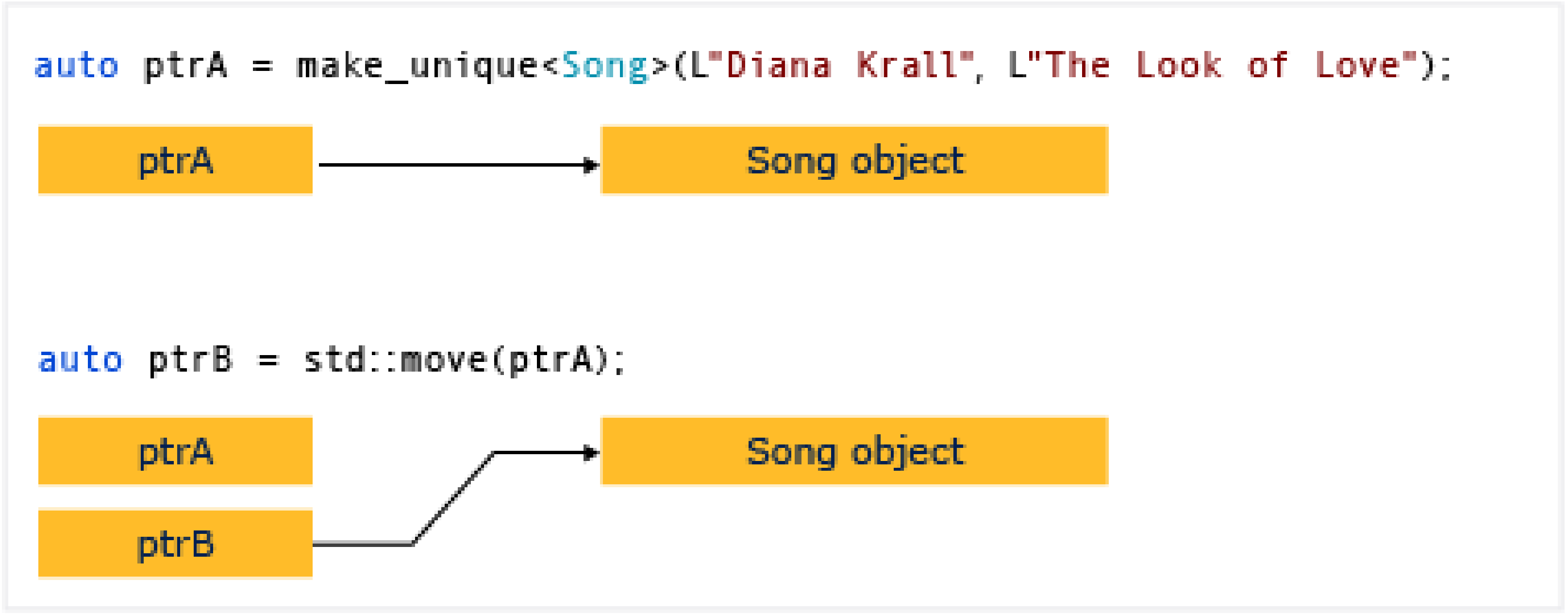
- *Shared Pointer*: Implemented using reference counting to keep track of how many “things” point to the object pointed to by the pointer. When this count goes to 0, the object is automatically deleted.
- *Weak Pointer*: Helps deal with cyclic reference which arises when using Shared Pointer. If you have two objects pointed to by two shared pointers and there is an internal shared pointer pointing to each others shared pointer then there will be a cyclick reference and the object will not be deleted when shared pointers go out of scope. To solve this problem, change the internal member from a shared_ptr to weak_ptr .



Shared Pointer[sp]

Standardized Smart Pointers Examples

- *Unique Pointer*: Light weight smart pointer with exclusive ownership. Use when pointer points to unique objects without sharing the objects between the pointers.



```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```

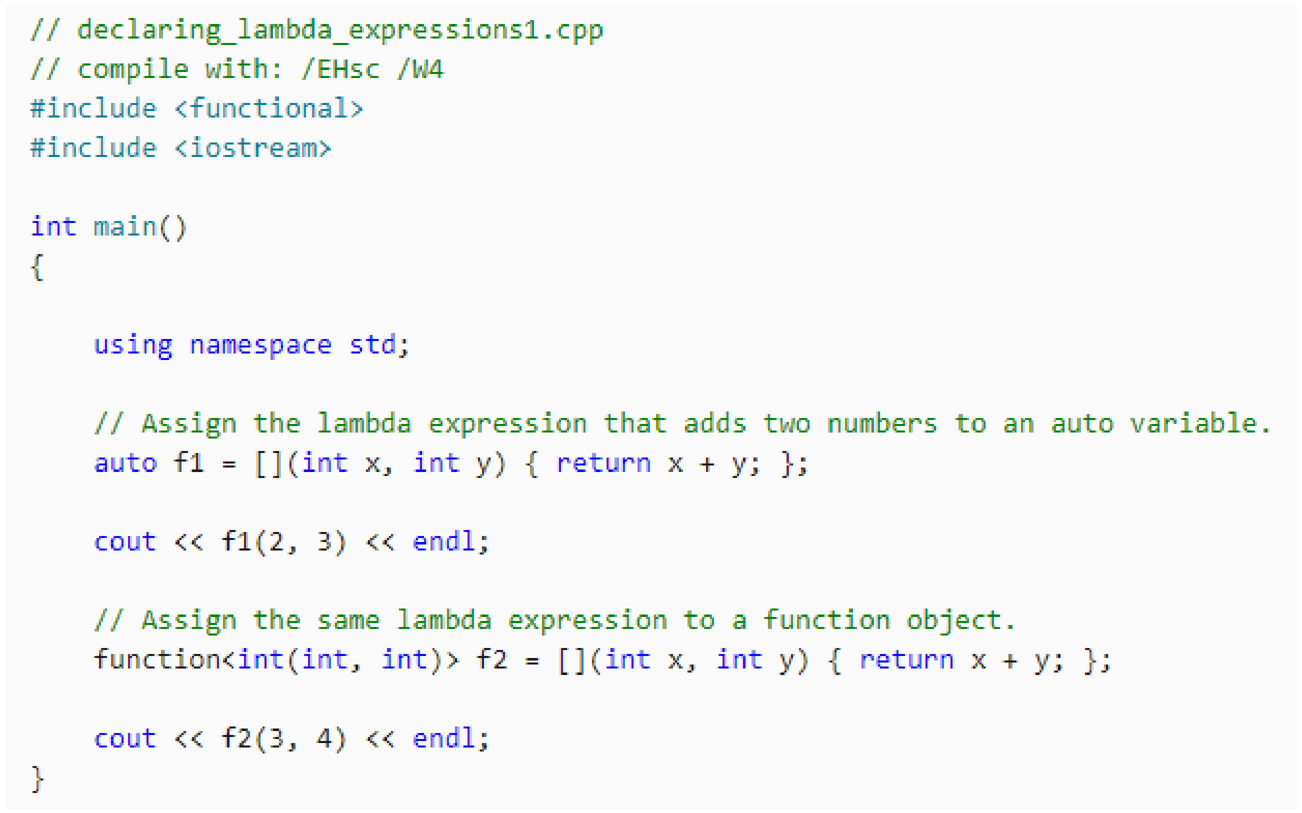
```
auto ptrB = std::move(ptrA);
```

Unique Pointer[up]

C++

Lambdas

- A lambda expression is a convenient way of defining an anonymous function object right at the location where it is invoked or passed as an argument to a function. The term lambda is borrowed from functional languages like Lisp and Scheme.
- Lambdas allow you to write the implementation of a functor inline, rather than having to declare a named function externally and pass it in.

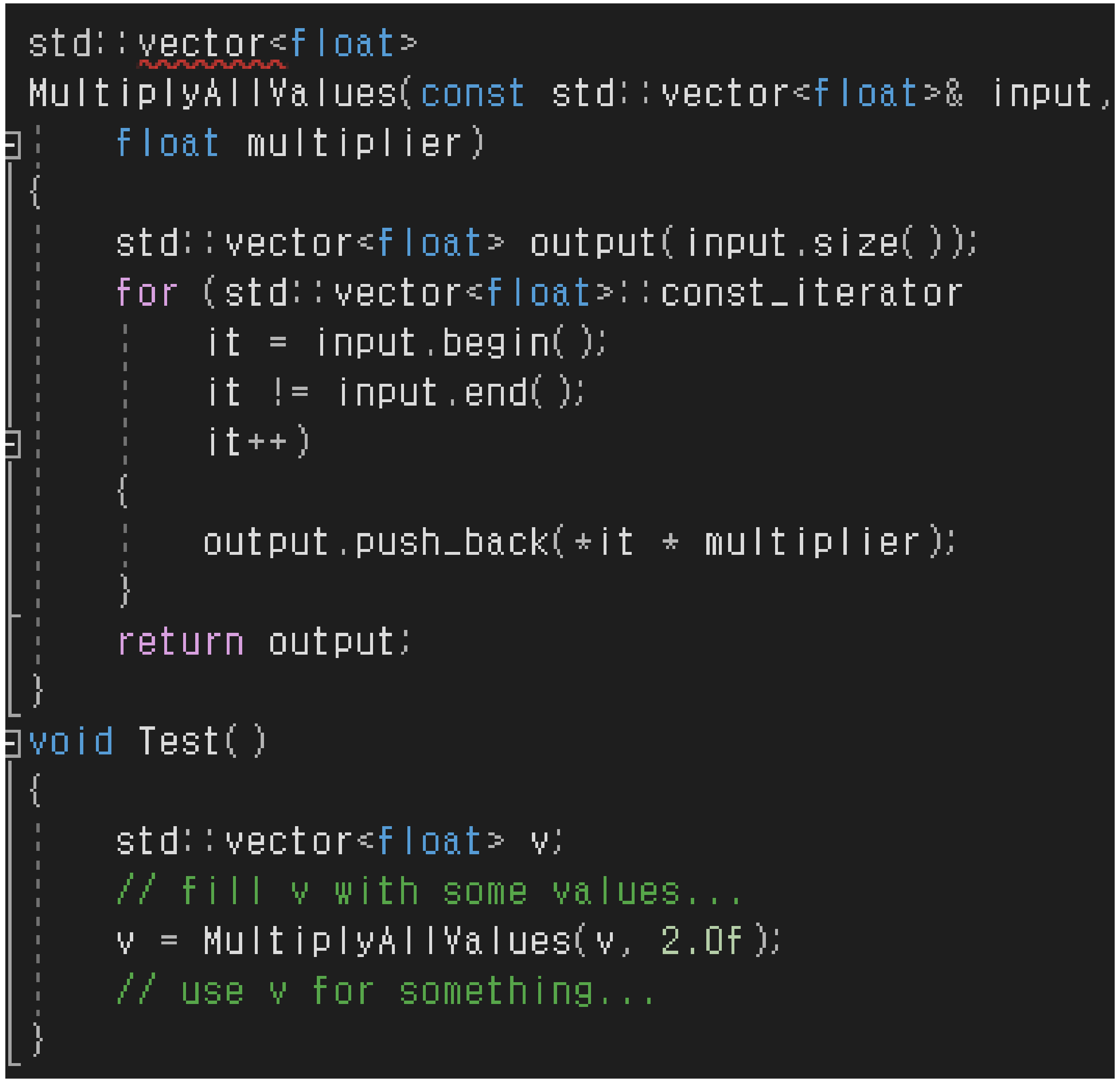


```cpp
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{

    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto variable.
    auto f1 = [](int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

C++

Move Semantics and Rvalue References

- One of the less-efficient aspects of the C++ language was the way it dealt with copying objects.
- As an example, consider a function that multiplies each value within a *std::vector* by a fixed multiplier and returns a new vector containing the results.
- In this example, unnecessary data copy is happens.
- Where?

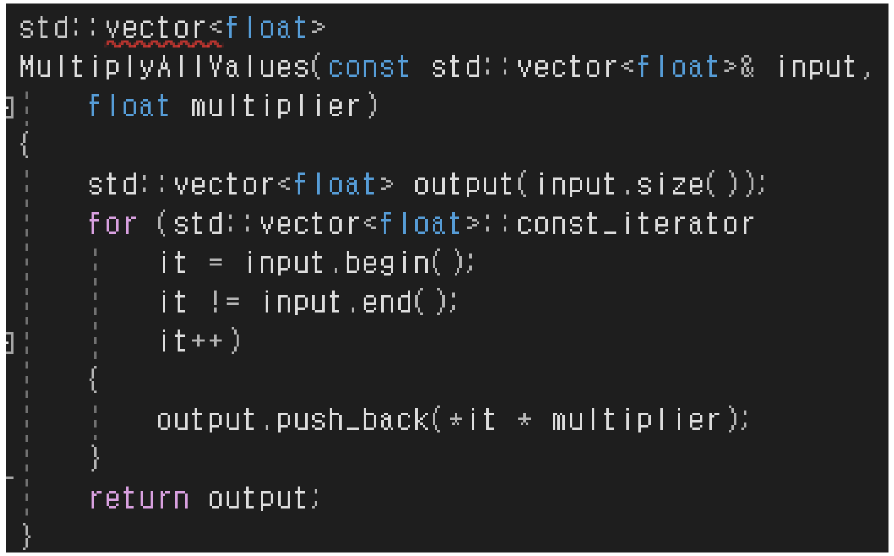


```
std::vector<float>
MultiplyAllValues(const std::vector<float>& input,
    float multiplier)
{
    std::vector<float> output(input.size());
    for (std::vector<float>::const_iterator
        it = input.begin();
        it != input.end();
        it++)
    {
        output.push_back(*it * multiplier);
    }
    return output;
}
void Test()
{
    std::vector<float> v;
    // fill v with some values...
    v = MultiplyAllValues(v, 2.0f);
    // use v for something...
}
```

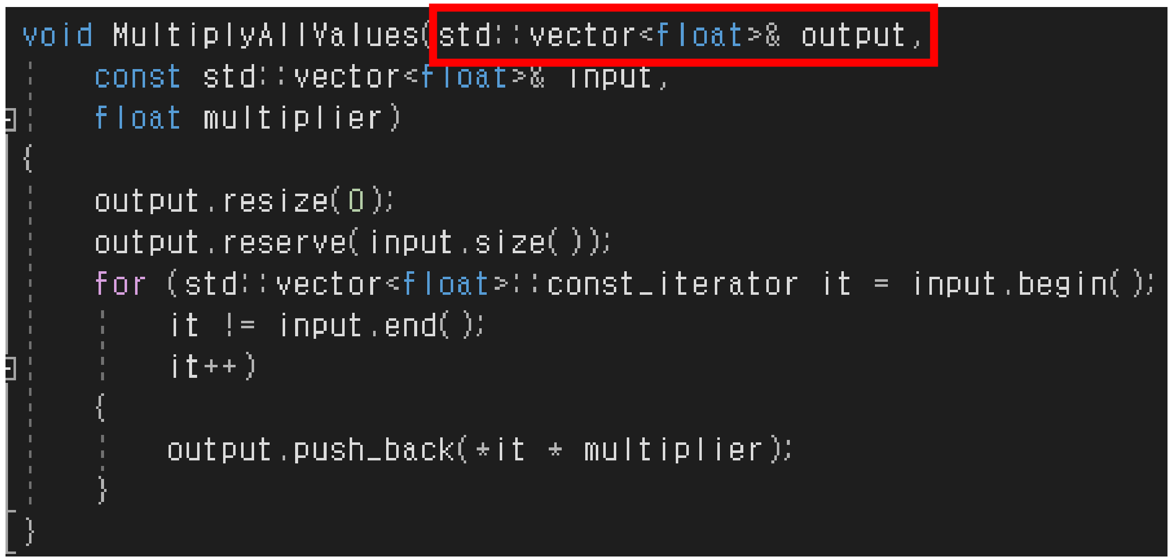
C++

Move Semantics and Rvalue References

- It happens when the return value is copied *back* into the vector *v*.
- Since the *source object* (the vector returned by the function) is a *temporary object*, it will be thrown away after being copied into *v*.
- To avoid such unnecessary copy, we can rewrite the function as follows:



```
std::vector<float>
MultiplyAllValues(const std::vector<float>& input,
    float multiplier)
{
    std::vector<float> output(input.size());
    for (std::vector<float>::const_iterator
        it = input.begin();
        it != input.end();
        it++)
    {
        output.push_back(*it * multiplier);
    }
    return output;
}
```



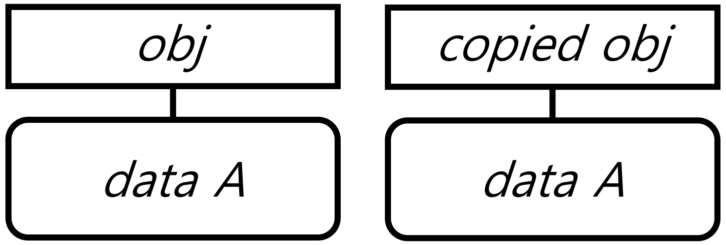
```
void MultiplyAllValues(std::vector<float>& output,
    const std::vector<float>& input,
    float multiplier)
{
    output.resize(0);
    output.reserve(input.size());
    for (std::vector<float>::const_iterator it = input.begin();
        it != input.end();
        it++)
    {
        output.push_back(*it * multiplier);
    }
}
```

Move Semantics and Rvalue References

- Recent C++ provides a mechanism that allows us to rectify these kinds of copying problems without having to change the function signature to pass the output object into the function by pointer or reference.
- This mechanism is known as *move semantics*, and it depends on being able to tell the difference between copying an *lvalue* object and copying an *rvalue* (temporary) object.
  - lvalue: This represents an object that occupies some identifiable location in memory (i.e. has an address).
  - rvalue: This represents an object that does not occupy some identifiable location in memory.

Move Semantics and Rvalue References

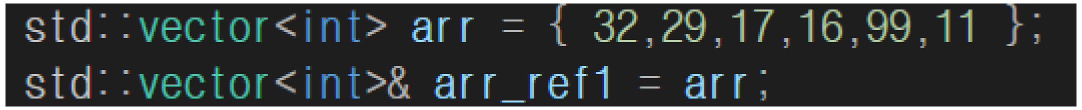
- Let’s assume we have an integer variable: ‘int var = 10;’
  - Then, the variable ‘var’ is an *lvalue* while the literal ‘10’ is an *rvalue*.
  - We can’t do ‘int var; 10 = var;’ because *rvalue* ‘10’ does not have a specific memory location.
- In early version of C++, there was no way to handle copying of *rvalues* differently from copying *lvalues*. Therefore, the copy constructor and assignment operator had to assume the worst and treat everything like an *lvalue*.
  - In the case of copying a container object like a std::vector, the copy constructor and assignment operator would have to perform a deep copy - copying not only the container object itself but all of the data it contains.



Deep Copy

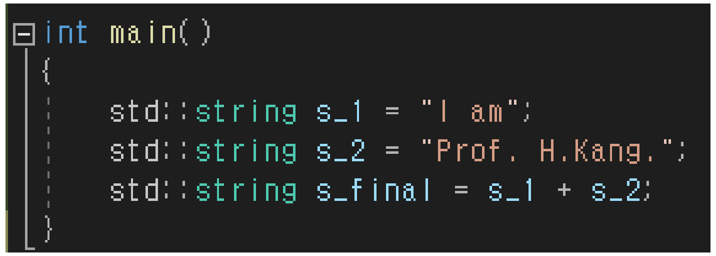
Rvalue References

- Rvalue references allow programmers to avoid logically unnecessary copying and to provide perfect forwarding functions.
- Let’s begin with an lvalue reference that is formed by placing an & after some type.



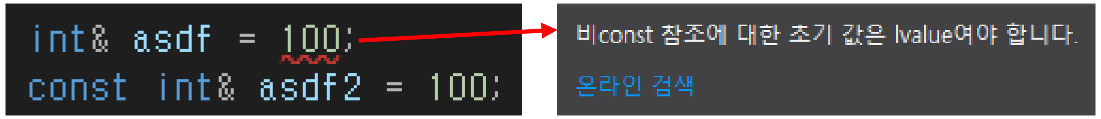
```
std::vector<int> arr = { 32,29,17,16,99,11 };
std::vector<int>& arr_ref1 = arr;
```

- In the following example, the compiler has to create a temporary object to hold the result of the + operator. Being a temporary one, the output is of course an *rvalue* that must be stored somewhere.



```
int main()
{
    std::string s_1 = "I am";
    std::string s_2 = "Prof. H.Kang.";
    std::string s_final = s_1 + s_2;
}
```

- The traditional C++ rules say that you are allowed to take the address of an *rvalue* only if you store it in a const (immutable) variable. More technically, you are allowed to bind a const *lvalue* to an *rvalue*.



```
int& asdf = 100;
const int& asdf2 = 100;
```

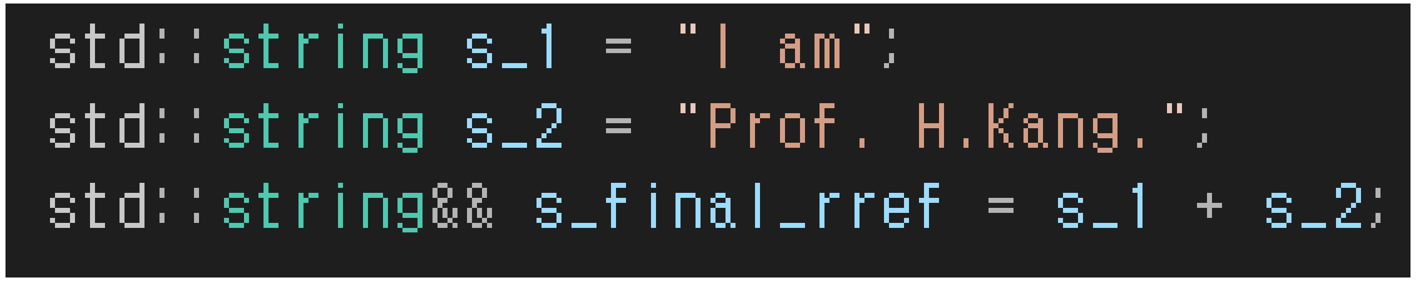
비const 참조에 대한 초기 값은 lvalue여야 합니다.

온라인 검색

C++

Rvalue References

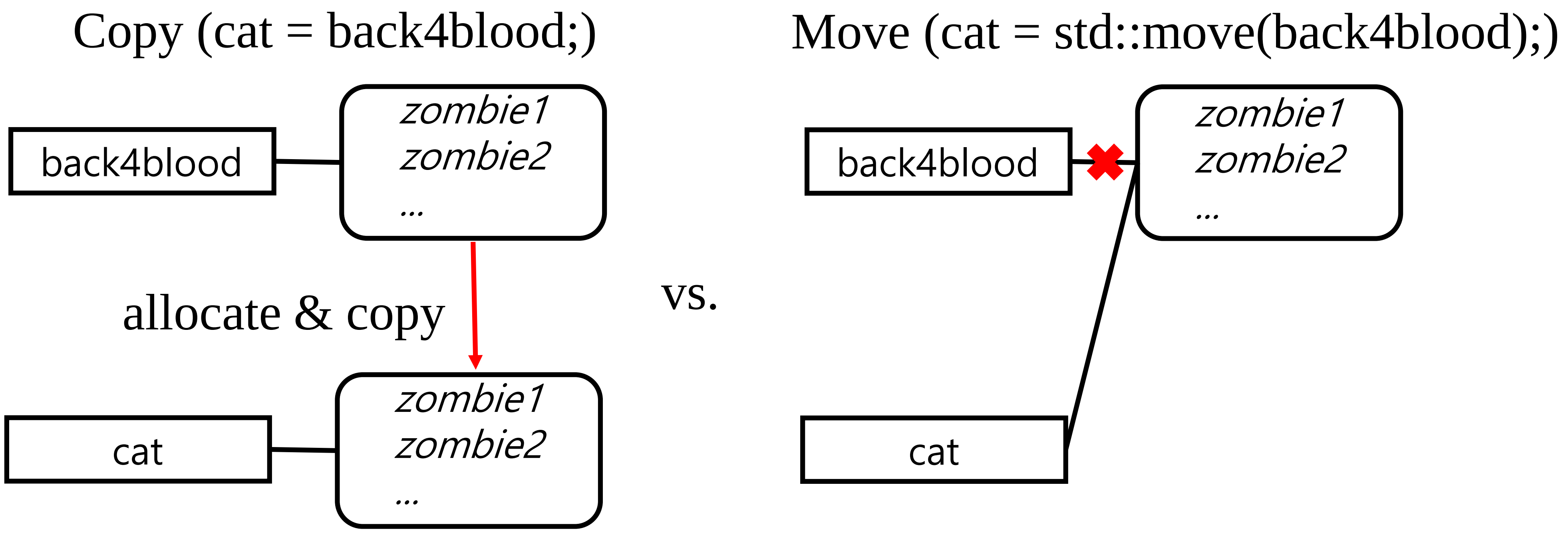
- C++11 has introduced a new type called rvalue reference, denoted by placing a double ampersand && after some type.
- In the following example, the *s_final_rref* is a reference to a temporary object, or an *rvalue* reference. There are not const around it, so we can modify the temporary string to our needs.



```cpp
std::string s_1 = "I am";
std::string s_2 = "Prof. H.Kang.";
std::string&& s_final_rref = s_1 + s_2;
```

Move semantics

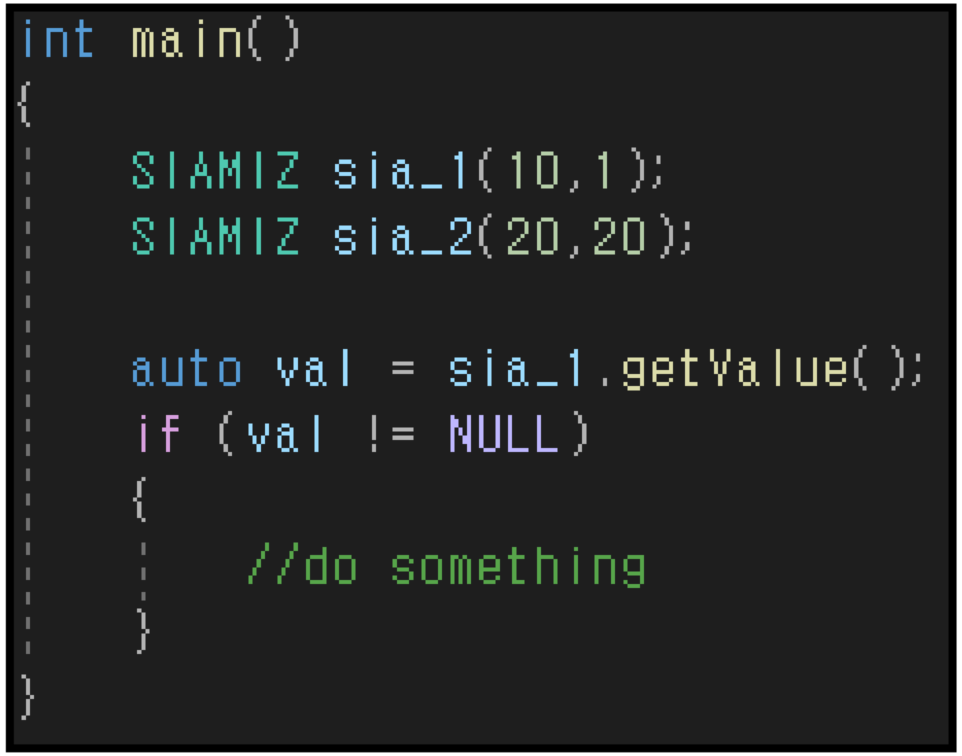
- Move semantics is a new way of moving resources around in an optimal way by avoiding unnecessary copies of temporary objects, based on *rvalue* references.
- When we copy an lvalue, we do a full deep copy as always. But when we copy an rvalue, we needn't perform a deep copy. Instead, we can simply steal the contents of the temporary object and move then directly into the destination object.



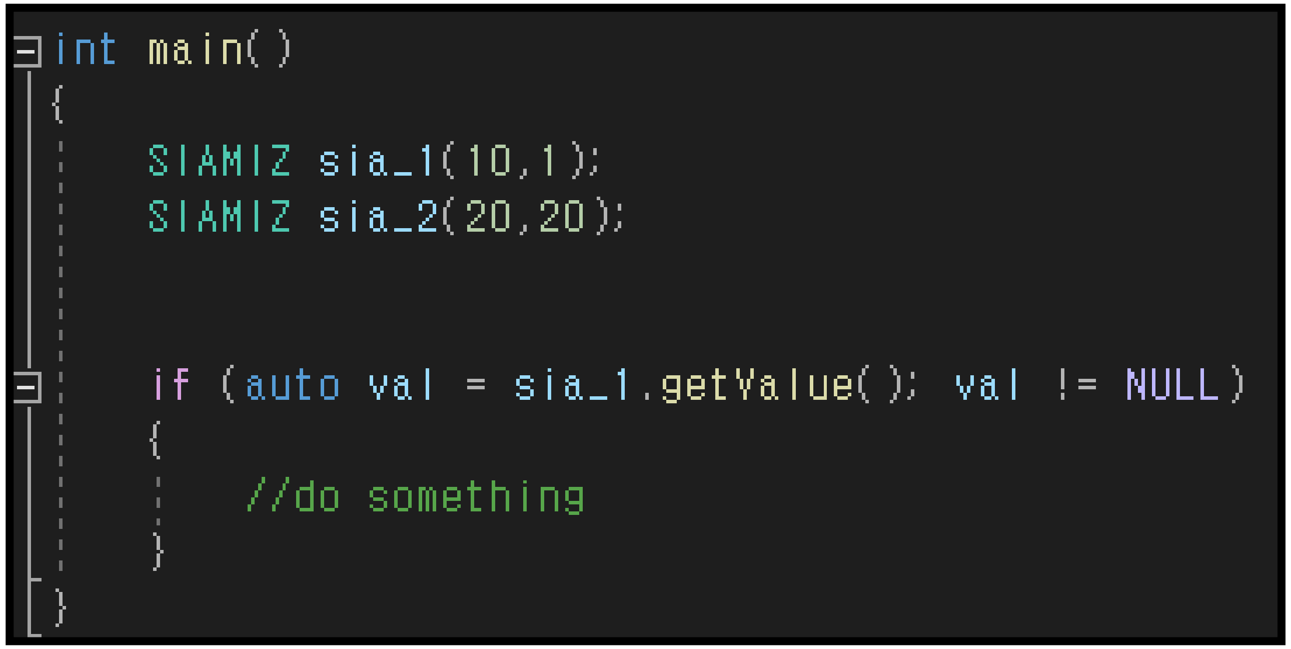
C++

If-Init statement

- Previously, the initializer is either declared before the statement and leaked into the ambient scope, or an explicit scope is used.
- With the new form, such code can be written more compactly, and the improved scope control makes erstwhile error-prone constructions a bit more robust.



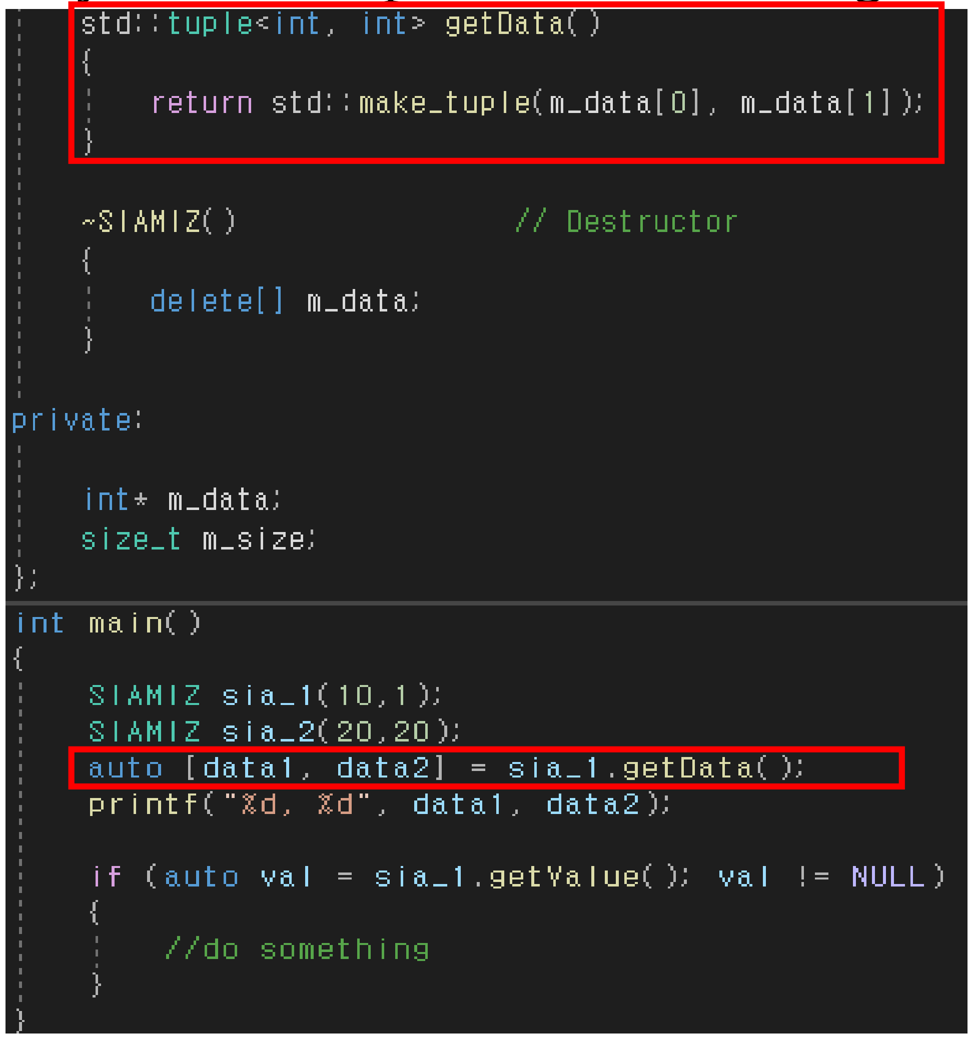
```
int main()
{
    SIAMIZ sia_1(10,1);
    SIAMIZ sia_2(20,20);

    auto val = sia_1.getValue();
    if (val != NULL)
    {
        //do something
    }
}
```



```
int main()
{
    SIAMIZ sia_1(10,1);
    SIAMIZ sia_2(20,20);


    if (auto val = sia_1.getValue(); val != NULL)
    {
        //do something
    }
}
```

Tuple

- A tuple is an object that packs elements of possibly different types together in a single object, just like pair objects do for pairs of elements, generalized



```
    std::tuple<int, int> getData()
    {
        return std::make_tuple(m_data[0], m_data[1]);
    }

    ~SIAMIZ()                   // Destructor
    {
        delete[] m_data;
    }

private:

    int* m_data;
    size_t m_size;
};
int main()
{
    SIAMIZ sia_1(10,1);
    SIAMIZ sia_2(20,20);
    auto [data1, data2] = sia_1.getData();
    printf("%d, %d", data1, data2);

    if (auto val = sia_1.getValue(); val != NULL)
    {
        //do something
    }
}
```

Numeric Representations

Numeric bases

- People think most naturally in *base ten*, also known as *decimal notation*.

- In this notation, ten distinct digits are used (0 through 9), and each digit from right to left represents the next highest power of 10.
  - $386 = (3 \times 10^3) + (8 \times 10^2) + (6 \times 10)$

Numeric Representations

Numeric bases

- In computer science, mathematical quantities such as integers and real valued numbers need to be stored in the computer’s memory. As we know, computers store numbers in *binary* format, meaning that only the two digits 0 and 1 are available. We call this a *base-two* representation.
  - 0b1011 = $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

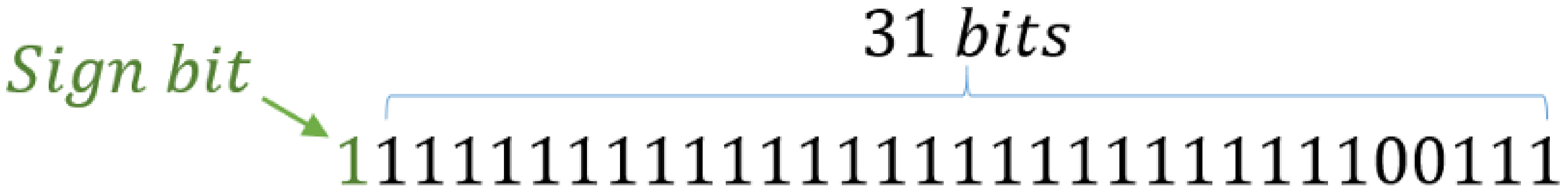
- Another common notation popular in computing circles is *hexadecimal*, or *base 16*. In this notation, the 10 digits 0 through 9 and the six letters A through F are used; the letters A through F replace the decimal values 10 through 15, respectively.
  - 0xA013 = $(10 \times 16^3) + (0 \times 16^2) + (1 \times 16^1) + (3 \times 16^0)$

Numeric Representations

Signed and unsigned integers

- The range of possible values for a 32-bit unsigned integer is 0x00000000 to 0xFFFFFFFF.

- To represent a signed integer in 32 bits, we need a way to differentiate between positive and negative values. One simple approach called the *sign* and *magnitude* encoding reserves the most significant bit as a *sign bit*.
  - When a sign bit is zero, the value is positive, and when it is one, the value is negative.
  - This leaves us 31 bits to represent the magnitude of the value, effectively cutting the range of possible magnitudes in half.



Numeric Representations

Signed and unsigned integers

- There are more efficient technique for encoding negative integers, called *two's complement* notation.
  - This notation has only one representation for the value zero, as opposed to the two representations possible with simple sign bit.
  - In 32-bit two's complement notation, the value 0xFFFFFFFF is interpreted to mean -1, and negative values count down from there.
  - So values from 0x00000000 to 0x7FFFFFFF represent positive integers, and 0x80000000 to 0xFFFFFFFF represent negative integers.

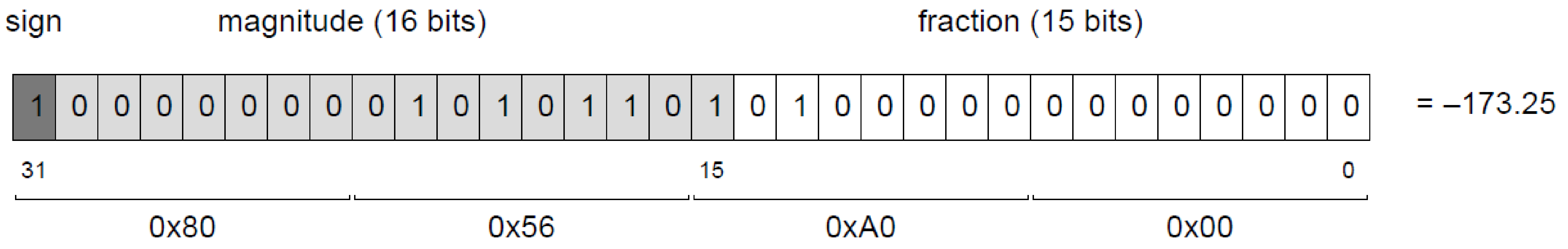
Two's complement binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Numeric Representations

Fixed-point notation

- Integers are great for representing whole numbers, but to represent fractions and irrational numbers we need a different format that expresses the concept of a decimal point.

- On early approach taken by computer scientists was to use fixed-point notation. In this notation, one arbitrarily chooses how many bits will be used to represent the whole part of the number, and the rest of the bits are used to represent the fractional part.

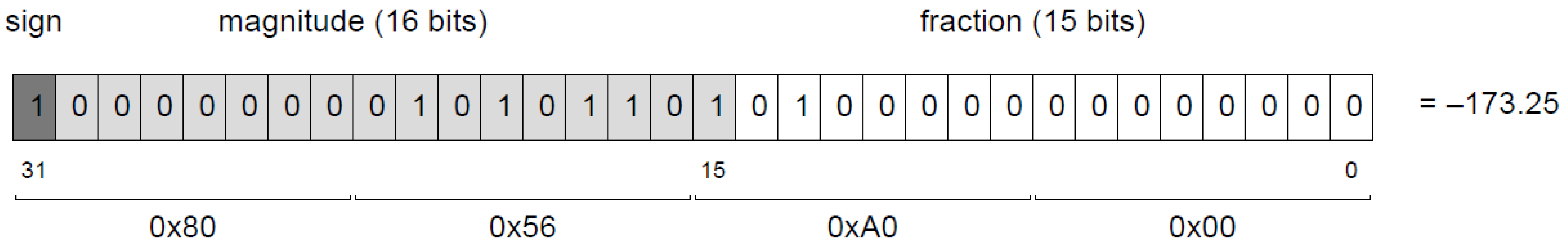


Numeric Representations

Fixed-point notation

- As we move from left to right, the magnitude bits represent decreasing powers of two (..., 16, 8, 4, 2, 1), while the fractional bits represent decreasing inverse powers of two($\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$, ...).

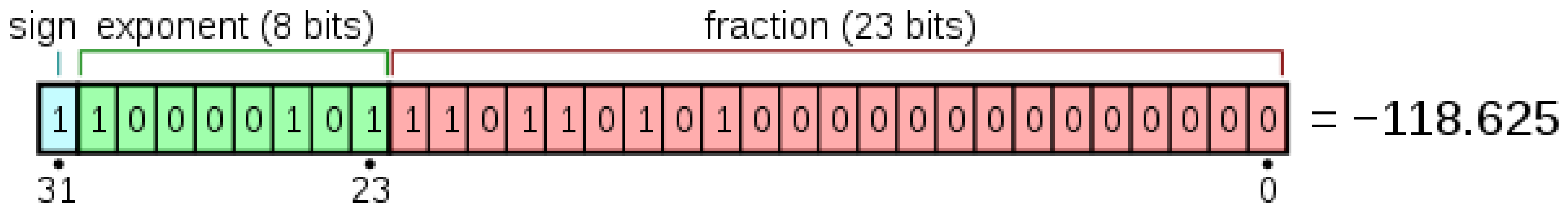
- The problem with fixed-point notation is that it constrains both the range of magnitudes that can be represented and the amount of precision we can achieve in the fractional part.



Numeric Representations

Floating-point notation

- In floating-point notation, the position of the decimal place is arbitrary and is specified with the help of an exponent.
- A floating-point number is broken into three parts
  - **Mantissa**: this contains the relevant digits of the number on both sides of the decimal point.
  - **Exponent**: this indicates where in that string of digits the decimal point.
  - **Sign bit**: this indicates whether the value is positive or negative.



Numeric Representations

Floating-point notation

- The value v represented by a sign bit s, an exponent e, and a mantissa m:
  - $v = s \times 2^{(e-127)} \times (1 + m)$.
- Example:
  - -118.625
  - a sign bit is 1 (negative number)
  - 118.625 = 1110110.101 (two’s complement)
  - $1110110.101 = 1.110110101 \times 2^{6}$, $e = 133$ $(6 + 127)$
  - Mantissa = 11011010100000000000000

Numeric Representations

Floating-point notation

- There are trade-off between magnitude and precision since there are a fixed number of bits in the mantissa.
- Let’s look at addition between 123456.7 and 0.009876543.
  - 123456.7 = 1.234567 × 10^5
  - 0.009876543 = 9.876543 × 10^{-3}

```
  e=5;  s=1.234567
+ e=-3; s=9.876543
```

```
  e=5;  s=1.234567
+ e=5;  s=0.00000009876543 (after shifting)
----------------------
  e=5;  s=1.23456709876543 (true sum)
  e=5;  s=1.234567         (after rounding and normalization)
```

Reference

- [lan] https://medium.com/@christianreyompad/other-variations-of-c-3a78634e7891
- [C++] https://stackoverflow.com/questions/3318898/why-is-c-that-powerful-concerning-game-development#:~:text=The%20main%20reason%20it's%20used,game%20development%20are%20C%2B%2B%20APIs.&text=Another%20trend%20worth%20noting%20is,much%20of%20the%20game%20logic.
- [comp] https://www.youtube.com/watch?v=jeg1haA3Eis&ab_channel=DataScientist
- [encap] https://www.softwaretestinghelp.com/encapsulation-in-cpp/
- [DDD] Martin, Robert C. (1997-03-09). "Java and C++: A critical comparison" (PDF). Objectmentor.com. Archived from the original (PDF) on 2005-10-24. Retrieved 2016-10-21.
- [abs] https://www.bogotobogo.com/DesignPatterns/abstractfactorymethod.php
- [auto] https://docs.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-160
- [null] https://en.cppreference.com/w/cpp/language/nullptr
- [for] https://www.geeksforgeeks.org/range-based-loop-c/
- [iter] https://en.cppreference.com/w/cpp/iterator/iterator
- [over] https://www.fluentcpp.com/2020/02/21/virtual-final-and-override-in-cpp/
- [final] https://en.cppreference.com/w/cpp/language/final
- [sp] https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-shared-ptr-instances?view=msvc-160
- [up] https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-160