UNIT-3—Dynamic Memory Management

Memory Managers

In C++, memory managers are responsible for allocating and deallocating memory dynamically during program execution.

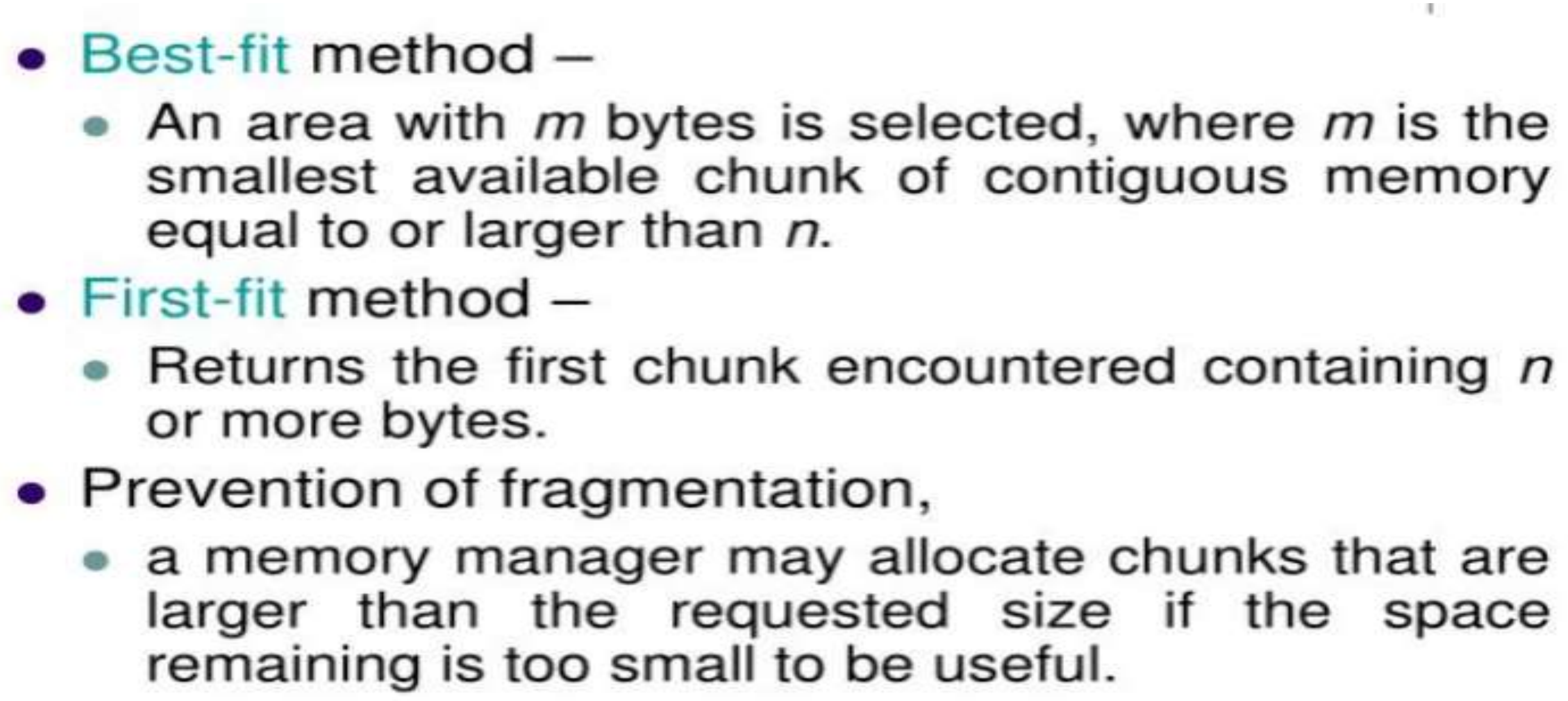
These memory management operations are crucial for the efficient use of system resources, and they play a key role in ensuring that memory is used effectively, particularly when dealing with large or complex applications.

C++ provides various mechanisms and tools for managing memory, including custom memory managers, standard memory management facilities (like new, delete, malloc, free), and smart pointers.

A memory manager can be seen as an abstraction that helps allocate memory (usually on the heap) and ensure proper deallocation, preventing issues such as memory leaks, dangling pointers, and double frees.

UNIT-3—Dynamic Memory Management

Memory Managers

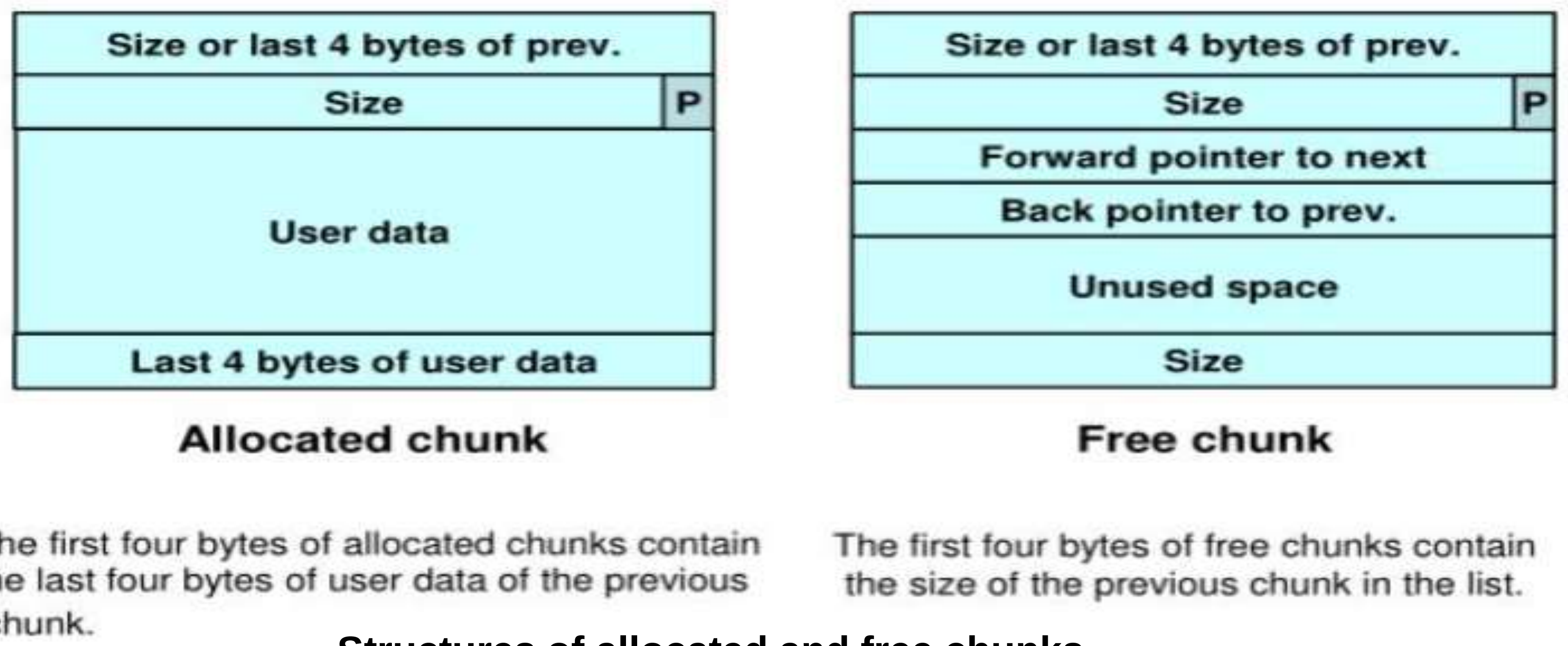


- Best-fit method –
  - An area with *m* bytes is selected, where *m* is the smallest available chunk of contiguous memory equal to or larger than *n*.
- First-fit method –
  - Returns the first chunk encountered containing *n* or more bytes.
- Prevention of fragmentation,
  - a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.

UNIT-3—Dynamic Memory Management

Doug Lea's Memory Allocator(dlmalloc) :

It is a native version of malloc. Doug Lea's malloc manages the heap and provides standard memory management. In dlmalloc memory chunks are either allocated to process or are free. In the figure P indicates PREV_INUSE bit to indicate whether or not free chunk allocated.



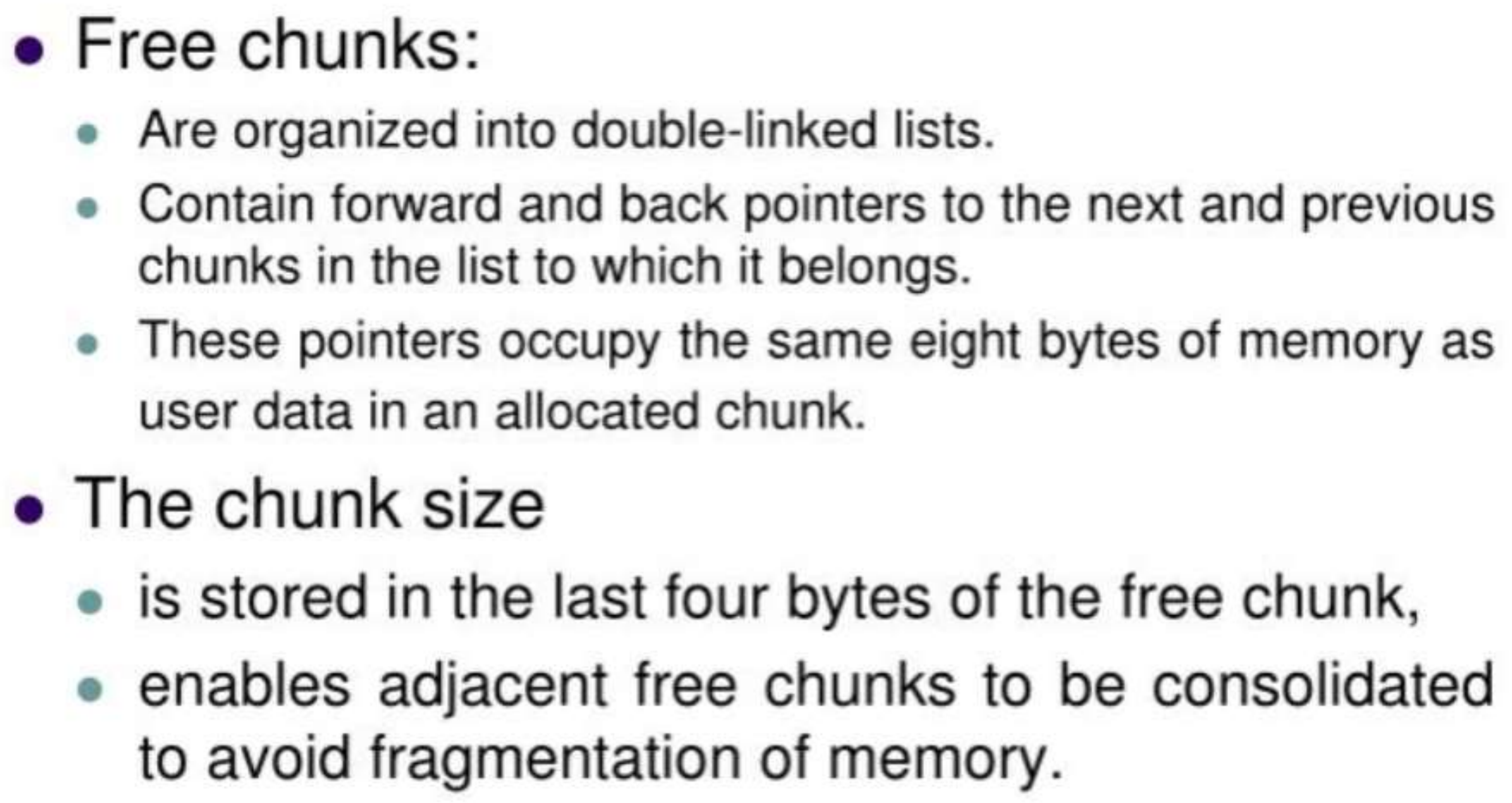
The first four bytes of allocated chunks contain the last four bytes of user data of the previous chunk.

The first four bytes of free chunks contain the size of the previous chunk in the list.

Structures of allocated and free chunks

UNIT-3—Dynamic Memory Management

Doug Lea’s Memory Allocator(dlmalloc) :



- Free chunks:
  - Are organized into double-linked lists.
  - Contain forward and back pointers to the next and previous chunks in the list to which it belongs.
  - These pointers occupy the same eight bytes of memory as user data in an allocated chunk.
- The chunk size
  - is stored in the last four bytes of the free chunk,
  - enables adjacent free chunks to be consolidated to avoid fragmentation of memory.

Structures of allocated and free chunks

UNIT-3—Dynamic Memory Management

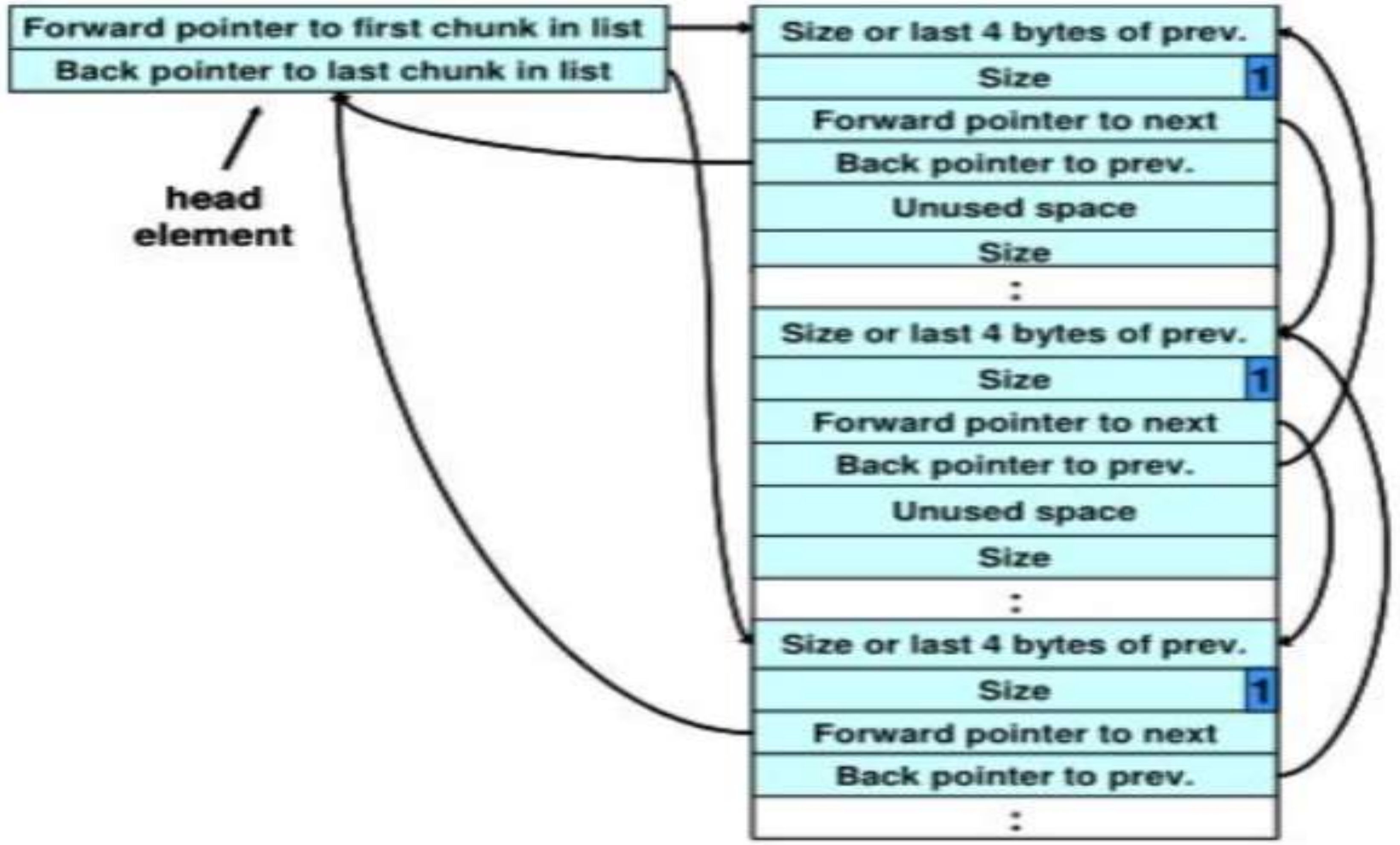
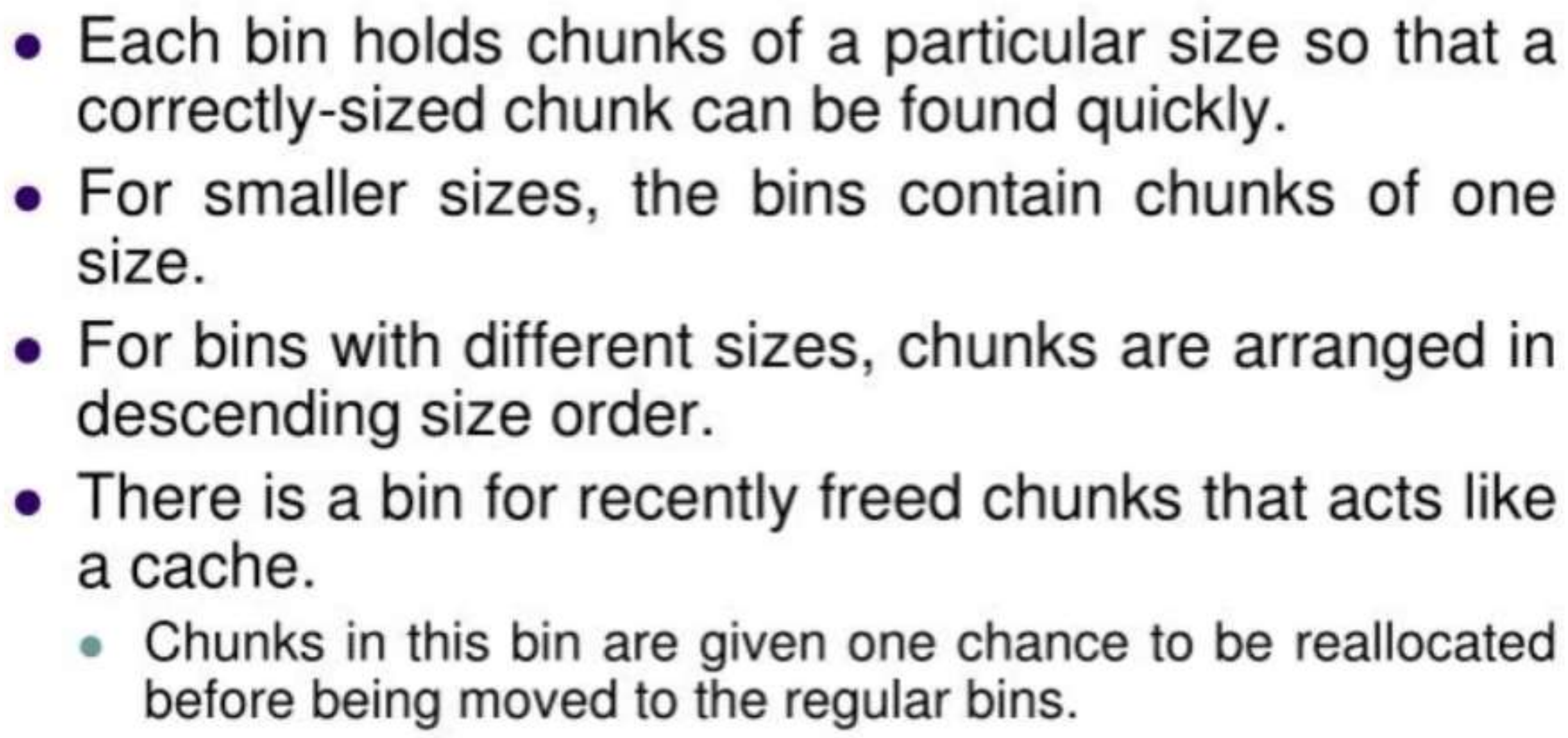


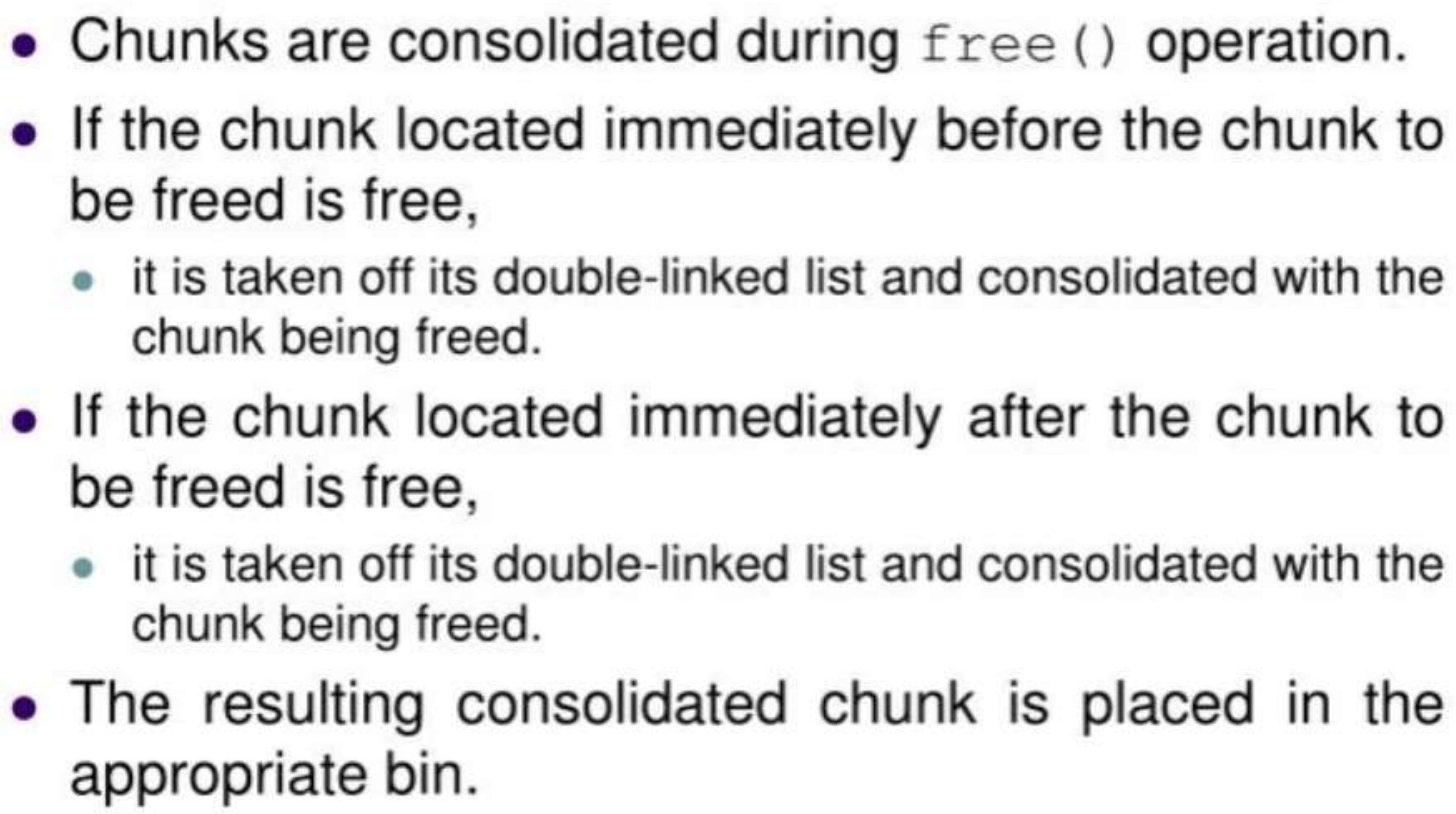
Fig: Free list double –linked structure

Doug Lea’s Memory Allocator(dlmalloc) :



- Each bin holds chunks of a particular size so that a correctly-sized chunk can be found quickly.
- For smaller sizes, the bins contain chunks of one size.
- For bins with different sizes, chunks are arranged in descending size order.
- There is a bin for recently freed chunks that acts like a cache.
  - Chunks in this bin are given one chance to be reallocated before being moved to the regular bins.

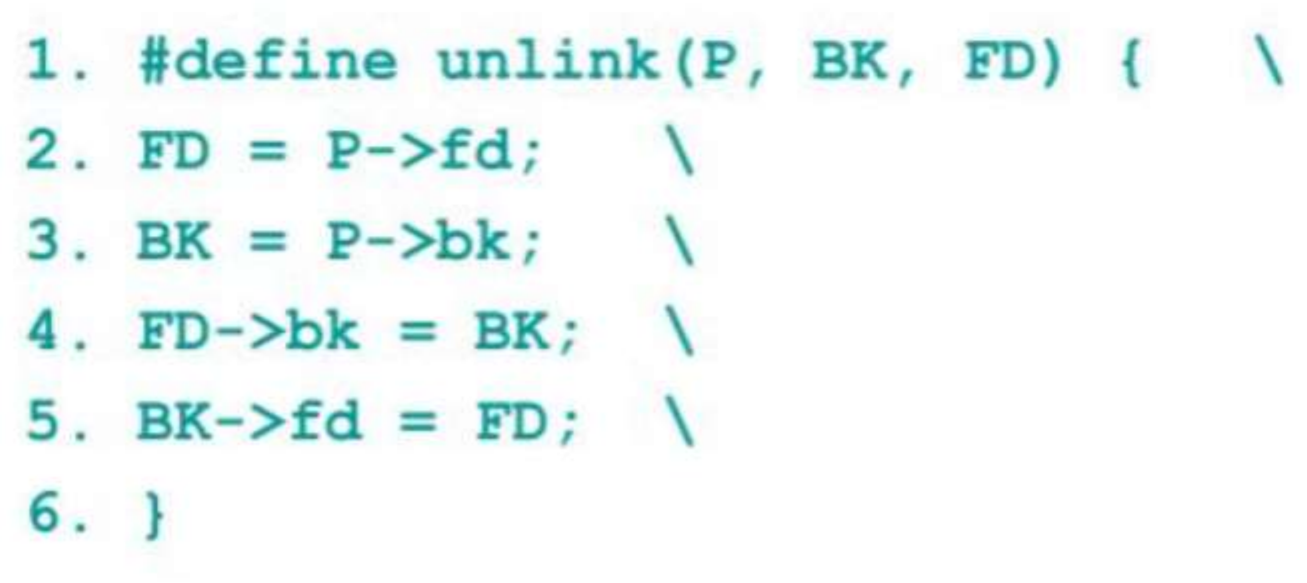
Doug Lea's Memory Allocator(dlmalloc) :



- Chunks are consolidated during `free()` operation.
- If the chunk located immediately before the chunk to be freed is free,
  - it is taken off its double-linked list and consolidated with the chunk being freed.
- If the chunk located immediately after the chunk to be freed is free,
  - it is taken off its double-linked list and consolidated with the chunk being freed.
- The resulting consolidated chunk is placed in the appropriate bin.

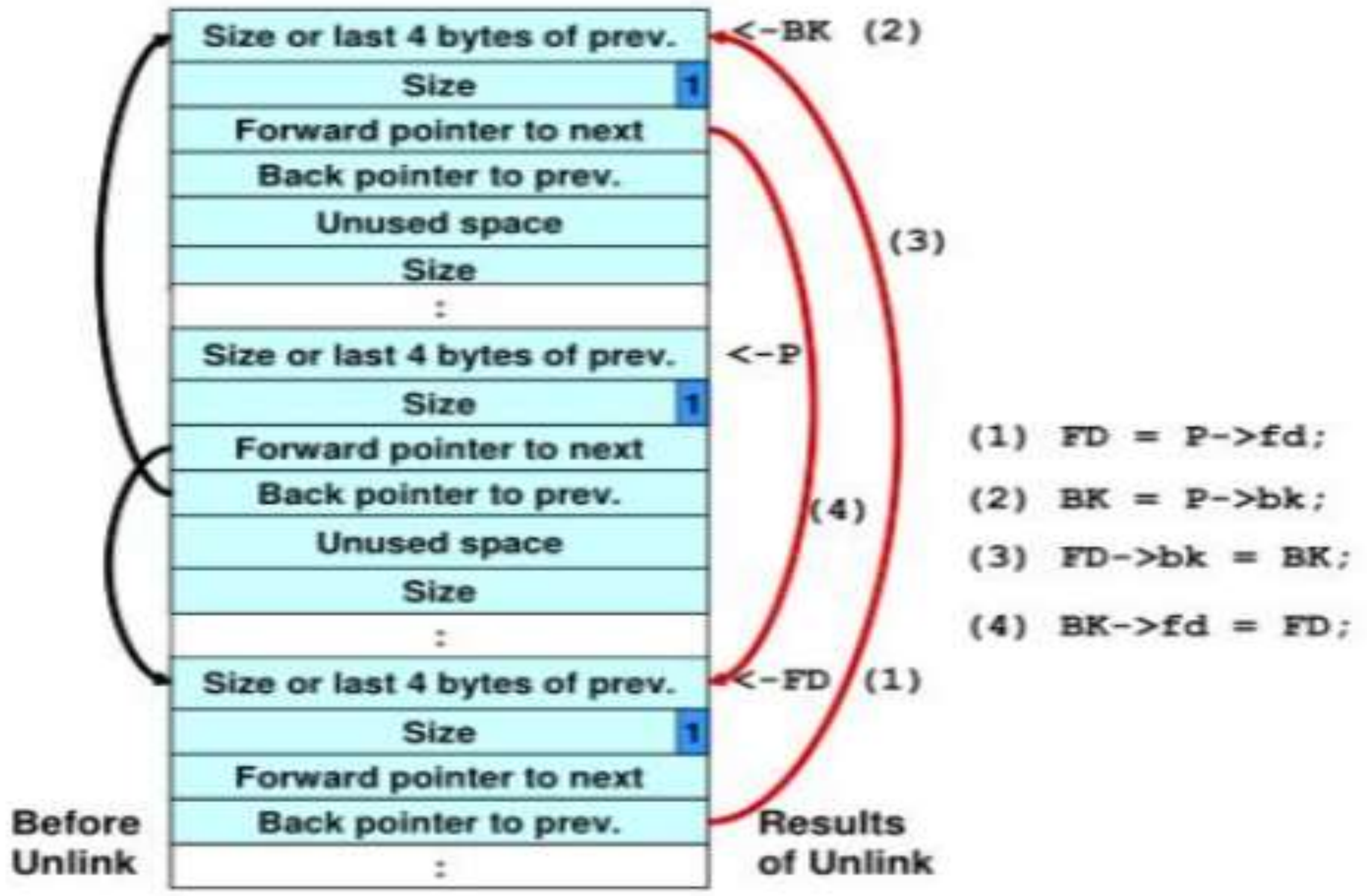
UNIT-3—Dynamic Memory Management

Doug Lea’s Memory Allocator(dlmalloc) :



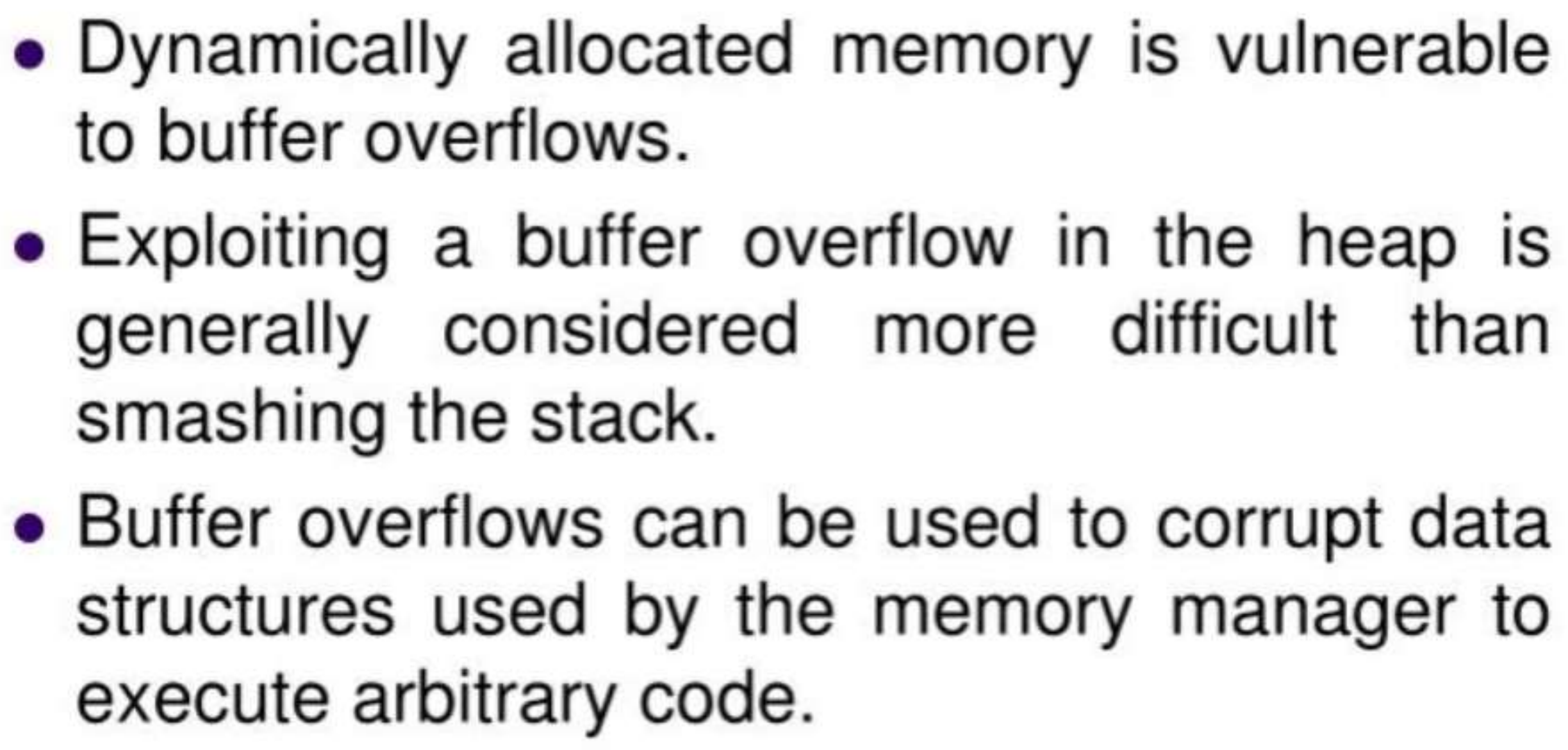
```
#define unlink(P, BK, FD) {   \
FD = P->fd;    \
BK = P->bk;    \
FD->bk = BK;   \
BK->fd = FD;   \
}
```

UNIT-3—Dynamic Memory Management

- 

```
(1) FD = P->fd;
(2) BK = P->bk;
(3) FD->bk = BK;
(4) BK->fd = FD;
```

Buffer Overflows on Heap



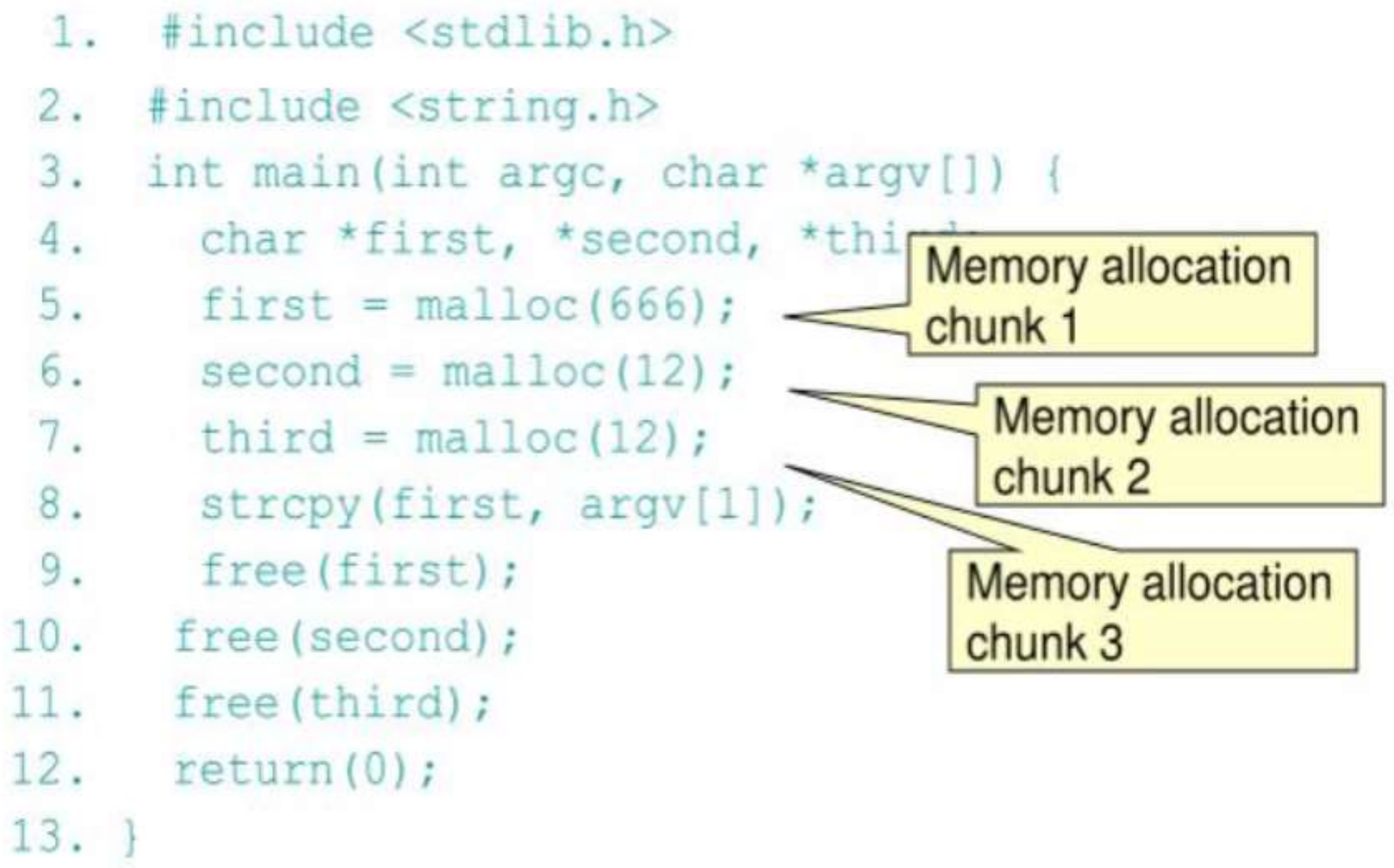
- Dynamically allocated memory is vulnerable to buffer overflows.
- Exploiting a buffer overflow in the heap is generally considered more difficult than smashing the stack.
- Buffer overflows can be used to corrupt data structures used by the memory manager to execute arbitrary code.

Unlink Technique:

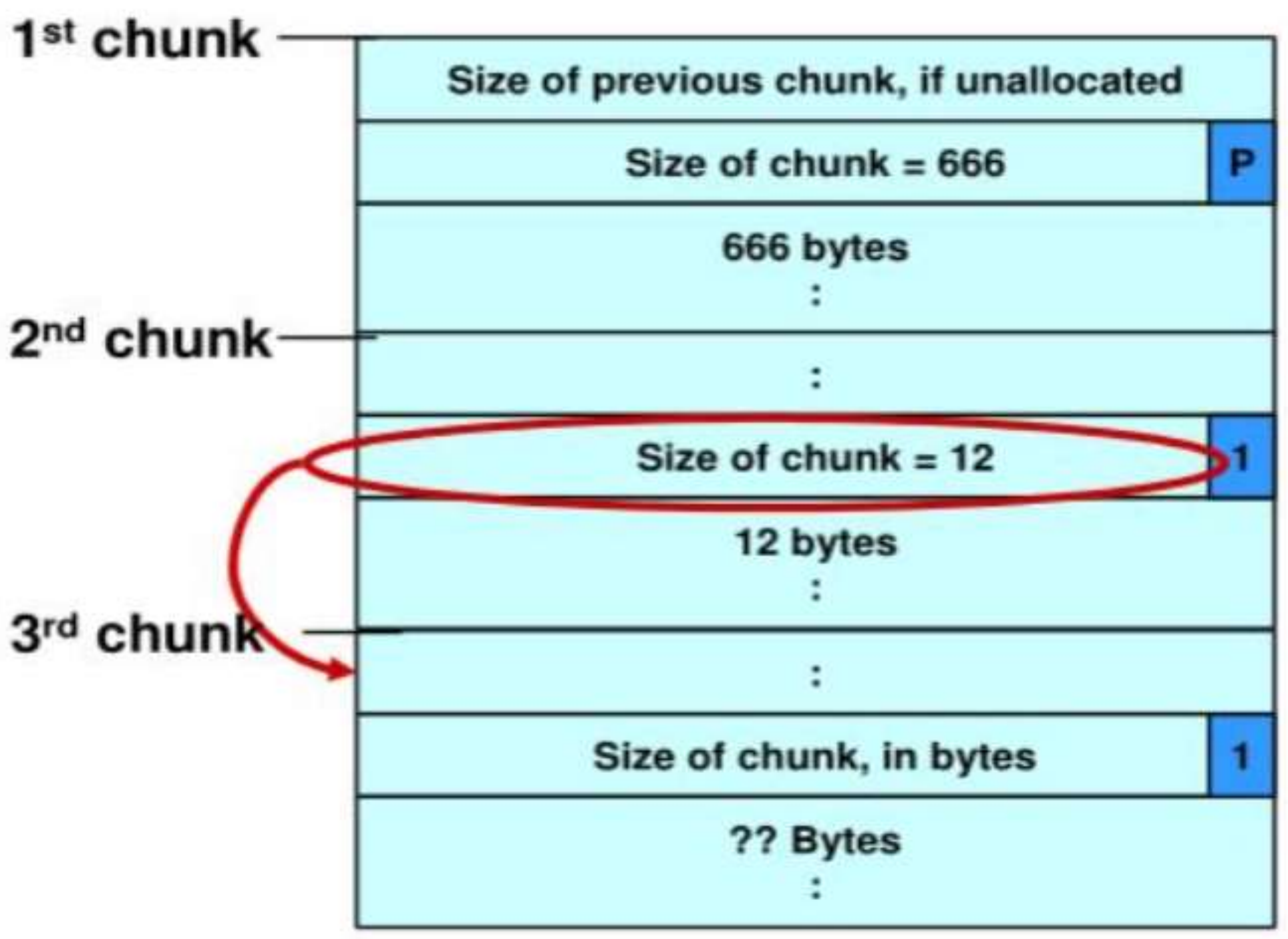
The unlink technique is used to exploit a buffer overflow to manipulate the boundary tags on chunks of memory to trick the unlink() macro into writing 4 bytes of data to an arbitrary location.

UNIT-3—Dynamic Memory Management

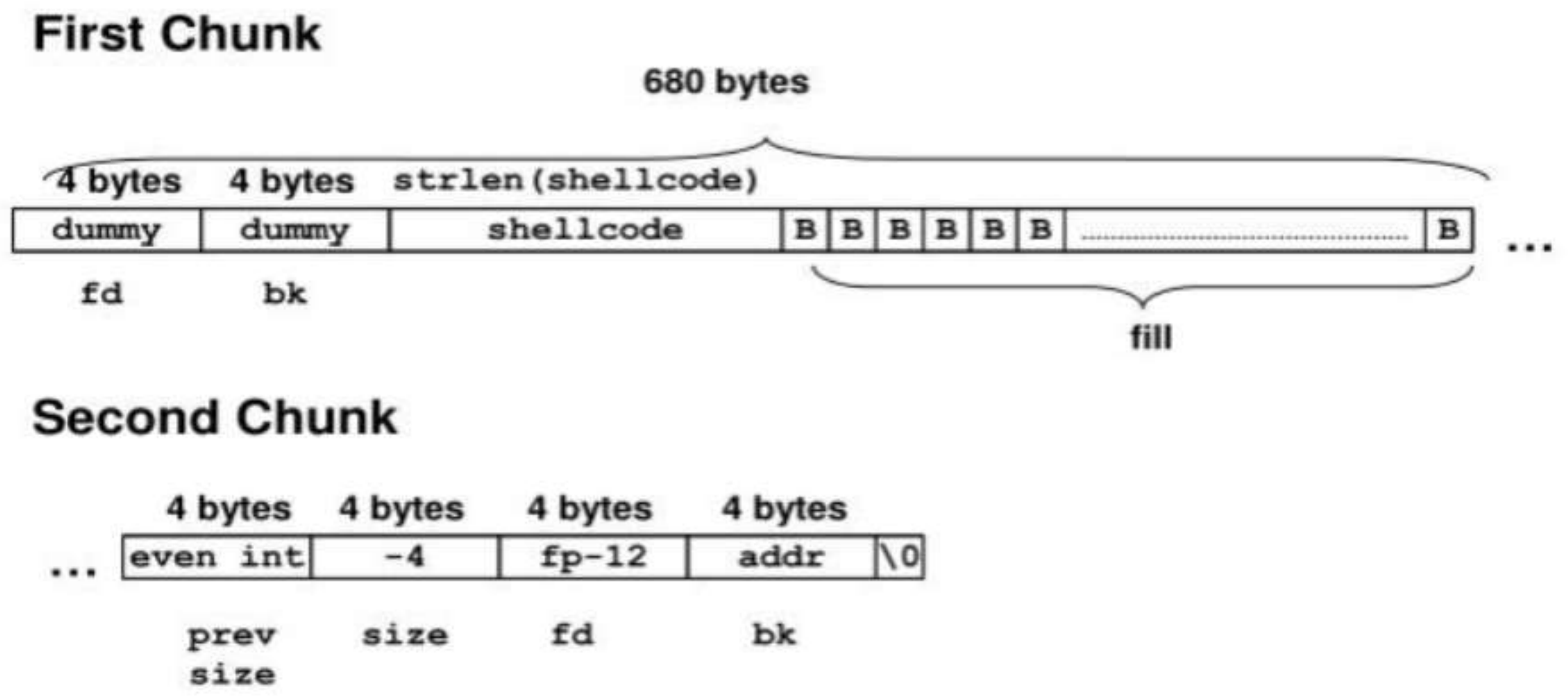
Unlink Technique:


```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
  char *first, *second, *thi
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);
  free(first);
  free(second);
  free(third);
  return(0);
}
```
Memory allocation chunk 1
Memory allocation chunk 2
Memory allocation chunk 3

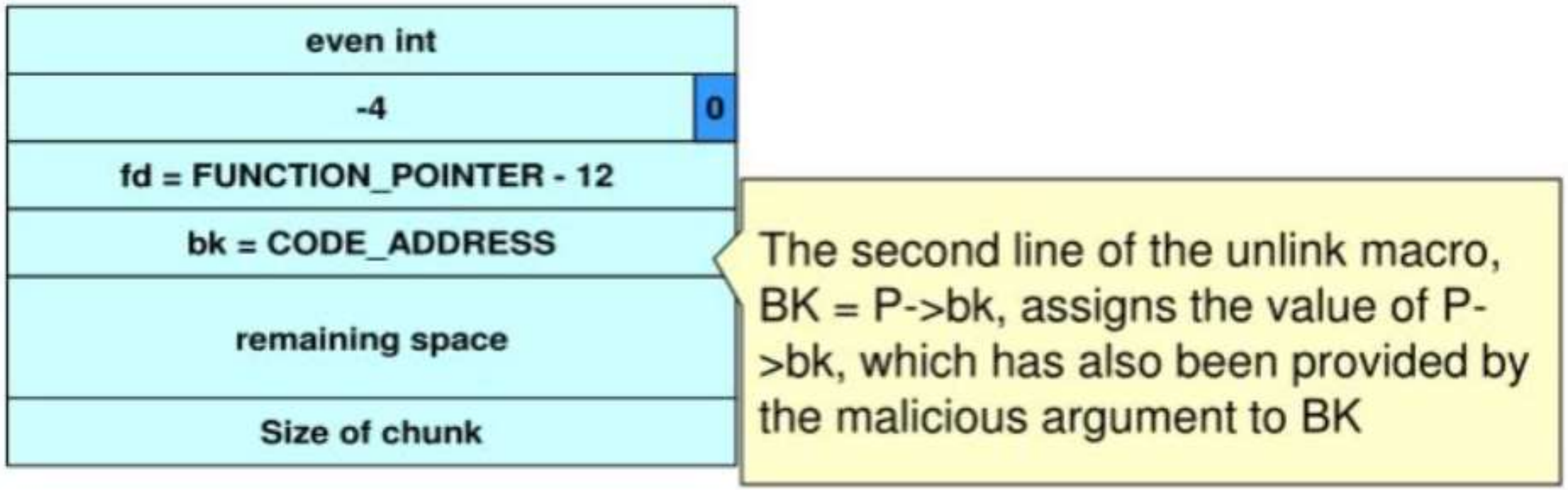
UNIT-3—Dynamic Memory Management



UNIT-3—Dynamic Memory Management

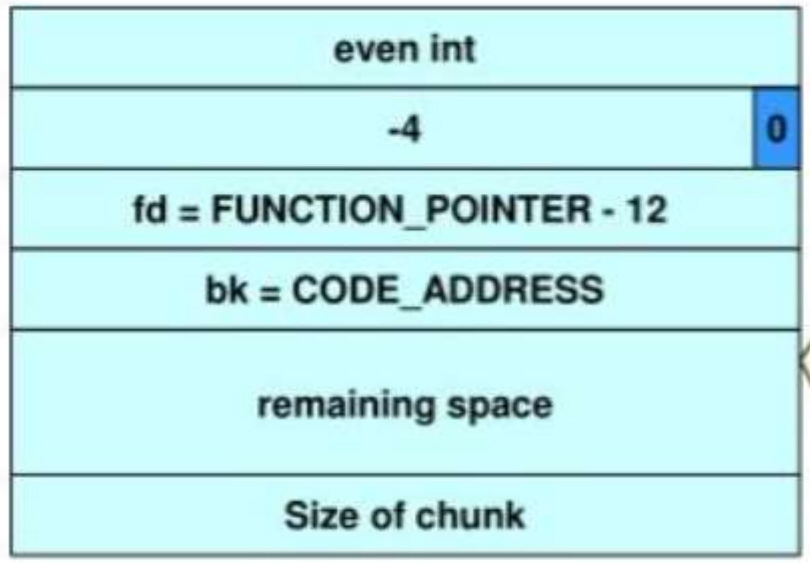


UNIT-2—Dynamic Memory Management



The second line of the unlink macro, BK = P->bk, assigns the value of P->bk, which has also been provided by the malicious argument to BK

UNIT-2—Dynamic Memory Management



The third line of the unlink() macro, FD->bk = BK, overwrites the address specified by FD + 12 (the offset of the bk field in the structure) with the value of BK

Mitigation Strategies

Null Pointers:

One obvious technique to reduce vulnerabilities in C and C++ programs is to set pointers to NULL after the referenced memory is deallocated.

Dangling pointers can result in writing to freed memory and double free vulnerabilities. Any attempt to dereference the pointer will result in a fault, which increases the likelihood that the error is detected during the implementation and test.

If the pointer is set to null memory can be freed multiple times without consequence.

Mitigation Strategies

Consistent Memory Management Conventions:
Following Conventions could be applied:

- Use the same pattern for allocating and freeing memory
- Allocate and free memory in the same module at the same level of abstraction
- Match allocations and deallocations

Mitigation Strategies

phkmalloc

PHKmalloc is a memory allocator designed for efficient and predictable memory management, especially for high-performance applications. It focuses on reducing memory fragmentation, minimizing the time spent on allocation/deallocation, and optimizing cache usage.

Key Features of PHKmalloc:

- Efficient Block Management: PHKmalloc reduces fragmentation by allocating memory in blocks and chunks of predefined sizes, which can be reused efficiently.
- Garbage Collection (Optional): In some configurations, PHKmalloc may include garbage collection mechanisms to prevent long-term fragmentation.
- Performance Optimizations: It minimizes overhead by reducing system calls and memory management-related locks.

Mitigation Strategies

Randomization:

Randomization works on the principle that it is harder to hit a moving target than a still target.

Randomizing the addresses of blocks of memory returned by the memory manager can make it more difficult to exploit heap based vulnerability.

Randomizing the memory addresses can occur in multiple locations. For both the windows and Unix Operating System, the memory manager requests the memory pages from the operating system, randomize both pages returned by the operating system.

Mitigation Strategies

OpenBSD:

OpenBSD Unix Variant was designed with a additional emphasis on security.

Open BSD adopted phkmalloc and adopted it to support randomization and guard pages(unmapped pages placed between all allocations of the memory the size of one page or larger to detect overflow.

Mitigation Strategies

jemalloc (short for "JavaScript Memory Allocator" originally, but now a general-purpose memory allocator) is a high-performance memory allocator used in various systems and applications, especially where memory fragmentation and efficiency are critical.

It was designed to address inefficiencies in traditional allocators (such as malloc and free in C/C++), offering better performance, especially for multi-threaded applications.

Several versions of jemalloc are available for FreeBSD, Linux, Windows and macOS, Mozilla Firefox

Notable Vulnerabilities

Many notable vulnerabilities result from the incorrect use of dynamic memory management.

CVS Buffer Overflow Vulnerability

Integer Security

Integer security refers to the practice of securing programs or systems that deal with integer values, particularly focusing on vulnerabilities that arise when integers are improperly handled, validated, or processed.

In many programming languages, integers are widely used for a range of tasks such as indexing arrays, managing memory, performing arithmetic operations, and interacting with external systems.

However, when integers are not properly validated, they can lead to significant security vulnerabilities.

Integer Security

Integer-related vulnerabilities often occur due to the following issues:

Integer Overflow and Underflow: When an integer exceeds its maximum value or falls below its minimum value, causing unexpected behaviour.

Integer Truncation: When larger integers are truncated (cut off) due to type conversion, leading to potential errors or security flaws.

Signed/Unsigned Mismatches: When signed and unsigned integers are mixed up in calculations, leading to incorrect results and potential exploitation.

Improper Bounds Checking: When integer values are used to index arrays or buffers without proper range validation, leading to out-of-bounds access.

Integer Security

Common Integer Vulnerabilities

Integer Overflow:

An integer overflow happens when an arithmetic operation attempts to create a value that exceeds the range that can be represented by the integer type (e.g., an int or unsigned int).

This may result in wrapping around the number or undefined behavior.

```
unsigned char x = 255;
x += 1; // x will wrap around to 0, potentially causing unexpected behavior
```

Integer Security

Common Integer Vulnerabilities

Integer Underflow:

An integer underflow occurs when an operation tries to decrease an integer value beyond its minimum representable value.

Example: A signed integer that can hold values from -128 to 127 may underflow if decremented past -128.

```
int x = -128;
x -= 1; // x will wrap around to 127, which may not be intended
```

Integer Security

Common Integer Vulnerabilities

Signed/Unsigned Mismatches:

Mixing signed and unsigned integers in operations can lead to unpredictable results due to the way negative values are handled in signed integers.

```
int a = -1;
unsigned int b = 1;
if (a < b) {  // This might not behave as expected due to signed/unsigned comparison
}
```

Integer Security

Common Integer Vulnerabilities

Integer Truncation:

When a larger integer type (e.g., long long) is truncated to fit into a smaller integer type (e.g., int), valuable data may be lost, causing potential errors.

```
long long large_num = 1234567890123456;
int small_num = (int)large_num; // Loss of data due to truncation
```

Integer Security

Common Integer Vulnerabilities

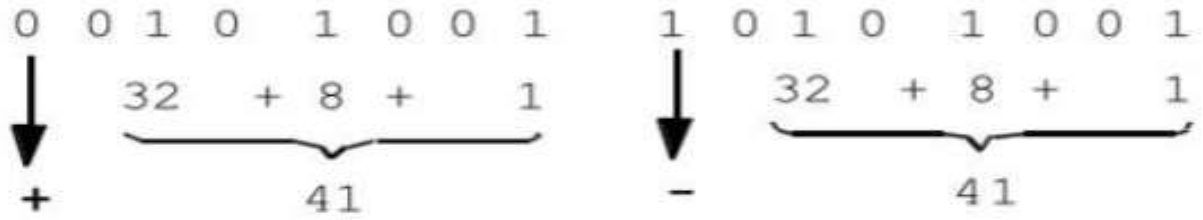
Improper Bound Checking:

Using integers to index into arrays or buffers without proper bounds checking can lead to buffer overflows or out-of-bounds memory access.

```
int arr[10];
int index = 20;  // index exceeds bounds of the array
arr[index] = 100; // This leads to out-of-bounds access
```

Integer Security-Signed Magnitude Representation

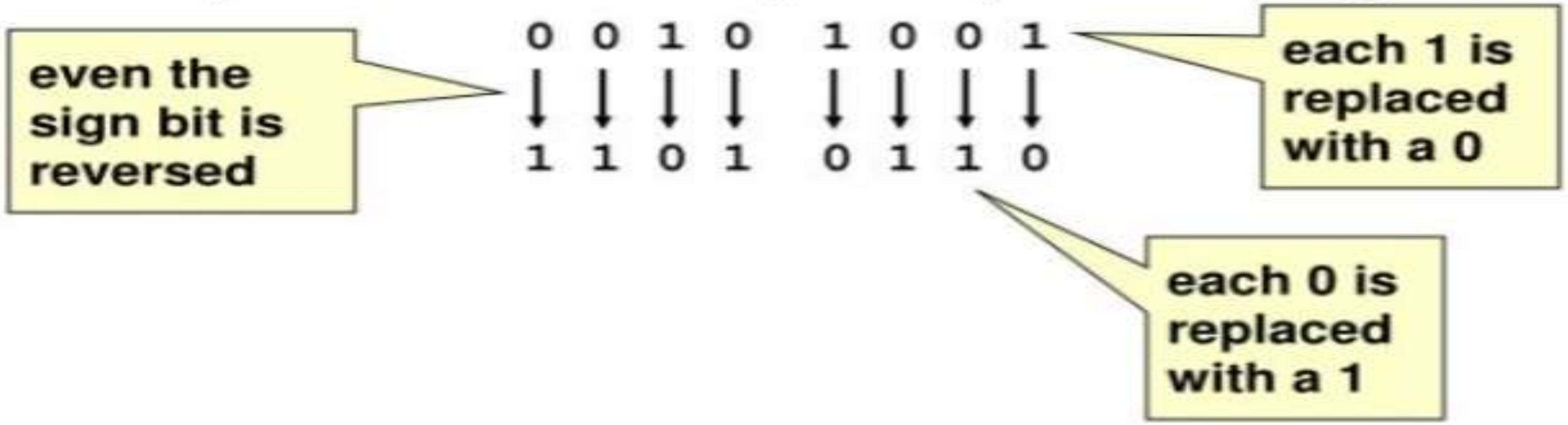
- Uses the high-order bit to indicate the sign
  - 0 for positive
  - 1 for negative
  - remaining low-order bits indicate the magnitude of the value



- Signed magnitude representation of +41 and -41

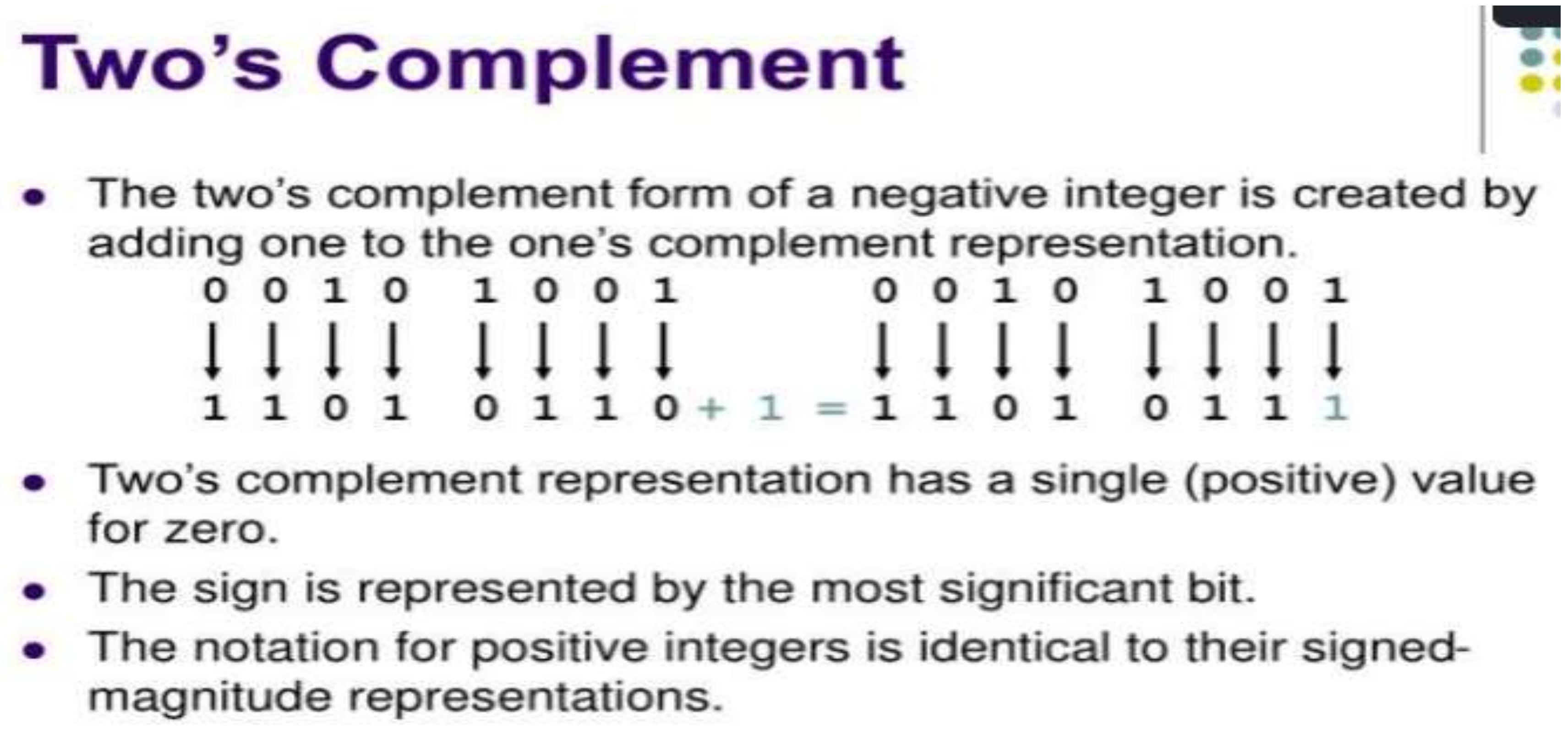
Integer Security

- One's complement replaced signed magnitude because the circuitry was too complicated.
- Negative numbers are represented in one's complement form by complementing each bit



even the sign bit is reversed
0 0 1 0 1 0 0 1
1 1 0 1 0 1 1 0
each 1 is replaced with a 0
each 0 is replaced with a 1

Integer Security



Two’s Complement

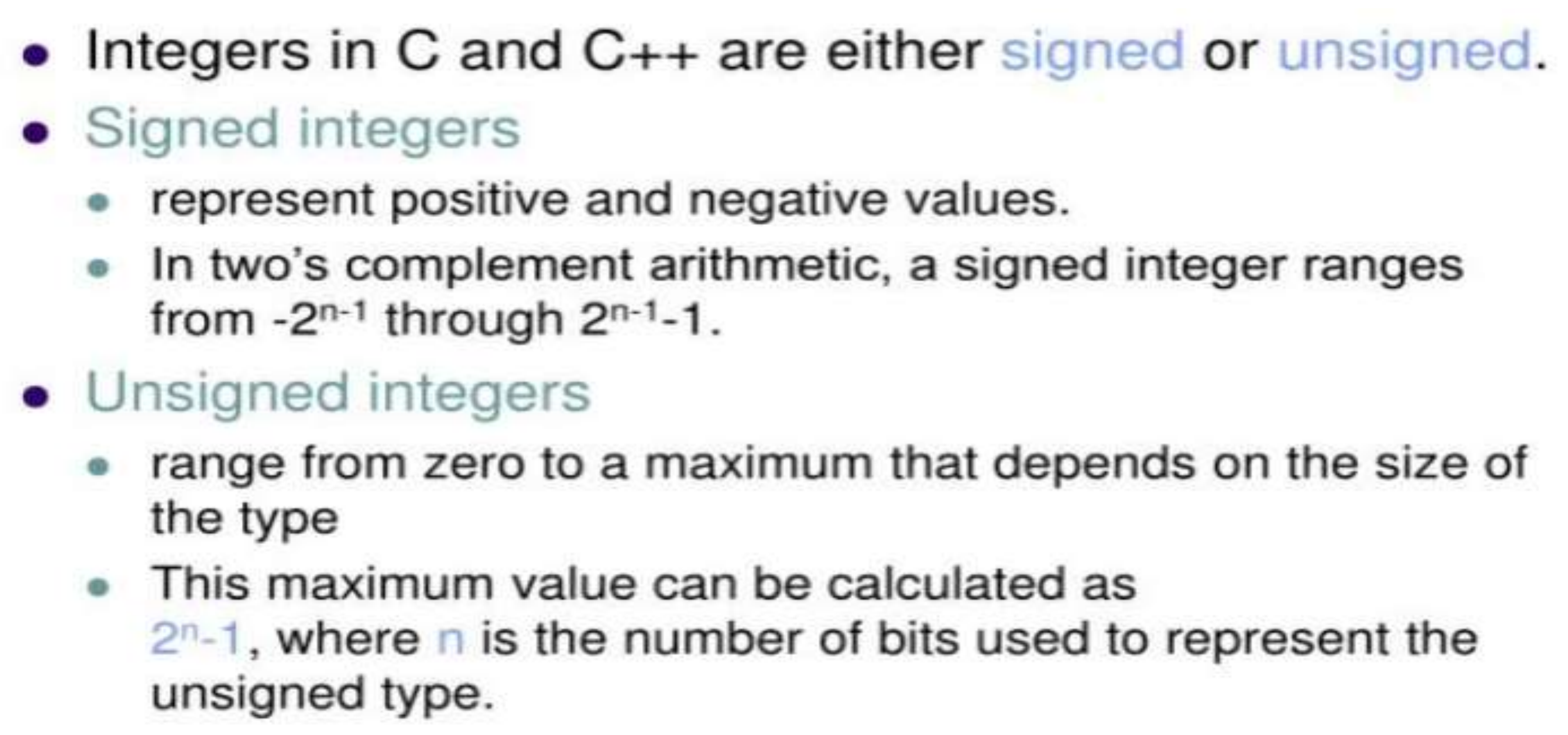
- The two’s complement form of a negative integer is created by adding one to the one’s complement representation.

```
0 0 1 0   1 0 0 1            0 0 1 0   1 0 0 1
↓ ↓ ↓ ↓   ↓ ↓ ↓ ↓            ↓ ↓ ↓ ↓   ↓ ↓ ↓ ↓
1 1 0 1   0 1 1 0 + 1 =      1 1 0 1   0 1 1 1
```

- Two’s complement representation has a single (positive) value for zero.
- The sign is represented by the most significant bit.
- The notation for positive integers is identical to their signed-magnitude representations.

Integer Security

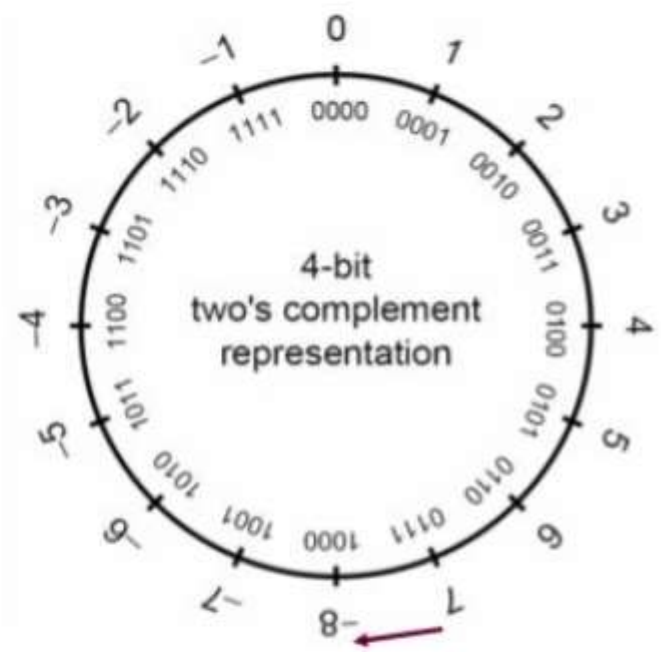
Signed and Unsigned Integers



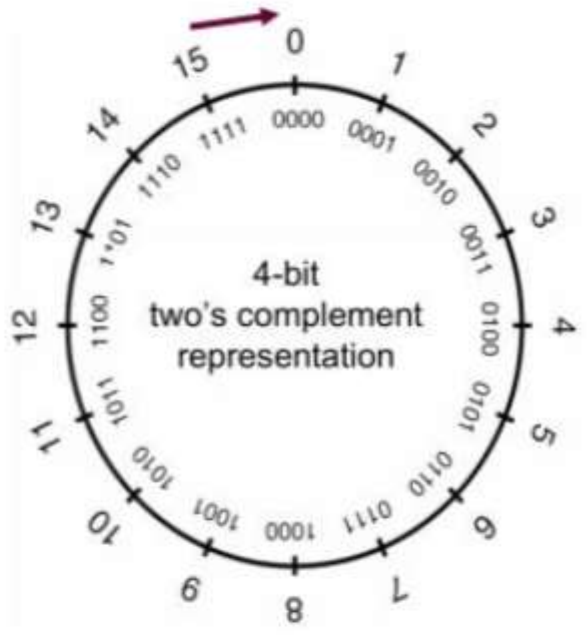
- Integers in C and C++ are either signed or unsigned.
- Signed integers
  - represent positive and negative values.
  - In two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$.
- Unsigned integers
  - range from zero to a maximum that depends on the size of the type
  - This maximum value can be calculated as 2^n-1, where n is the number of bits used to represent the unsigned type.

Integer Security

Signed and Unsigned Integers



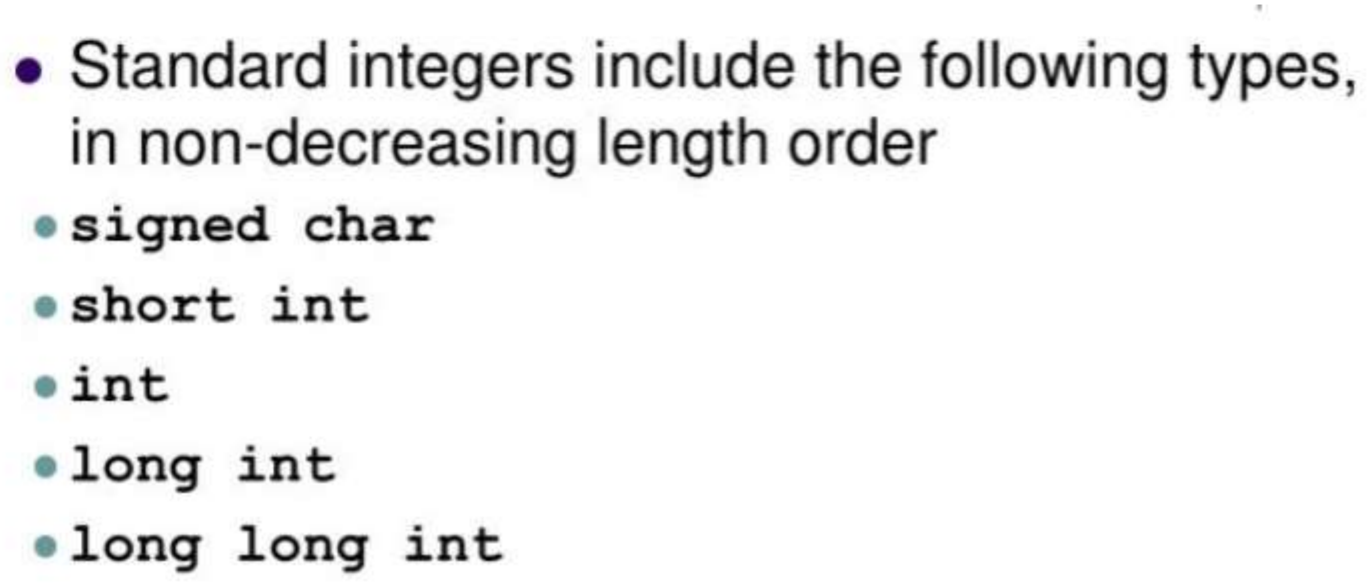
Signed Integer



Unsigned Integer

Integer Security

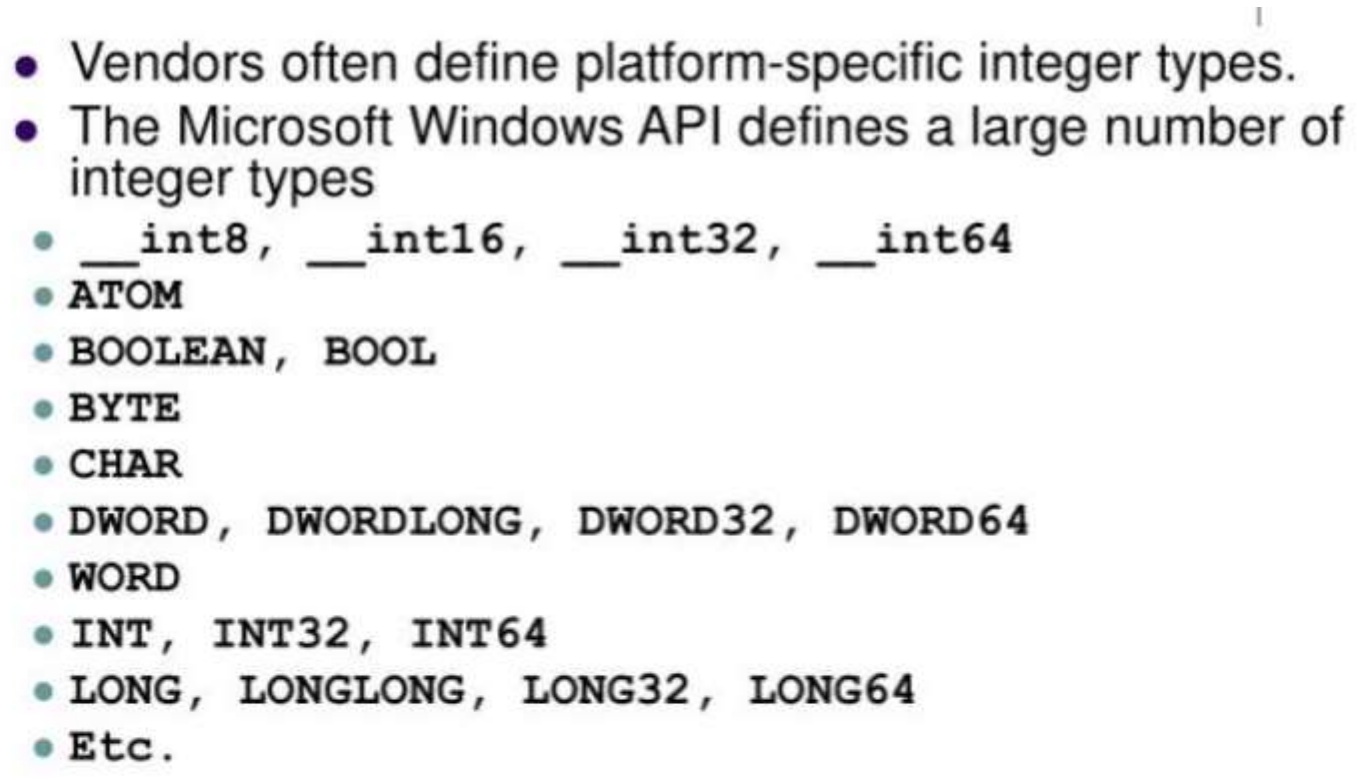
Standard Integer Types



- Standard integers include the following types, in non-decreasing length order
  - `signed char`
  - `short int`
  - `int`
  - `long int`
  - `long long int`

Integer Security

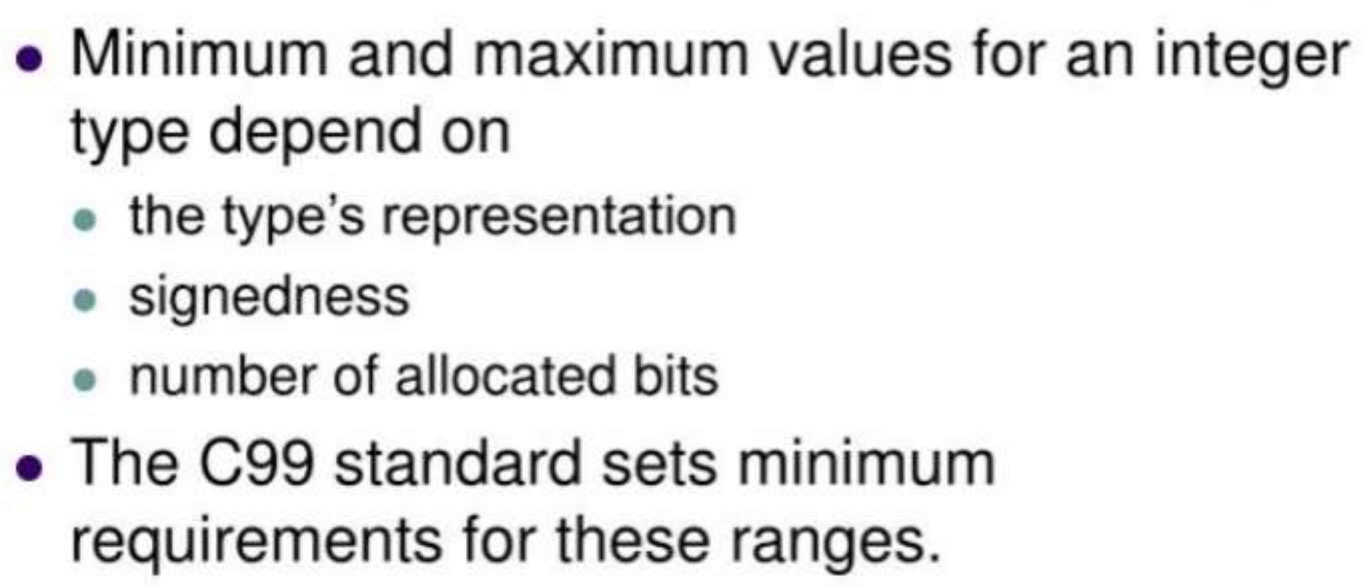
Platform Specific Integer types



- Vendors often define platform-specific integer types.
- The Microsoft Windows API defines a large number of integer types
  - `__int8, __int16, __int32, __int64`
  - `ATOM`
  - `BOOLEAN, BOOL`
  - `BYTE`
  - `CHAR`
  - `DWORD, DWORDLONG, DWORD32, DWORD64`
  - `WORD`
  - `INT, INT32, INT64`
  - `LONG, LONGLONG, LONG32, LONG64`
  - `Etc.`

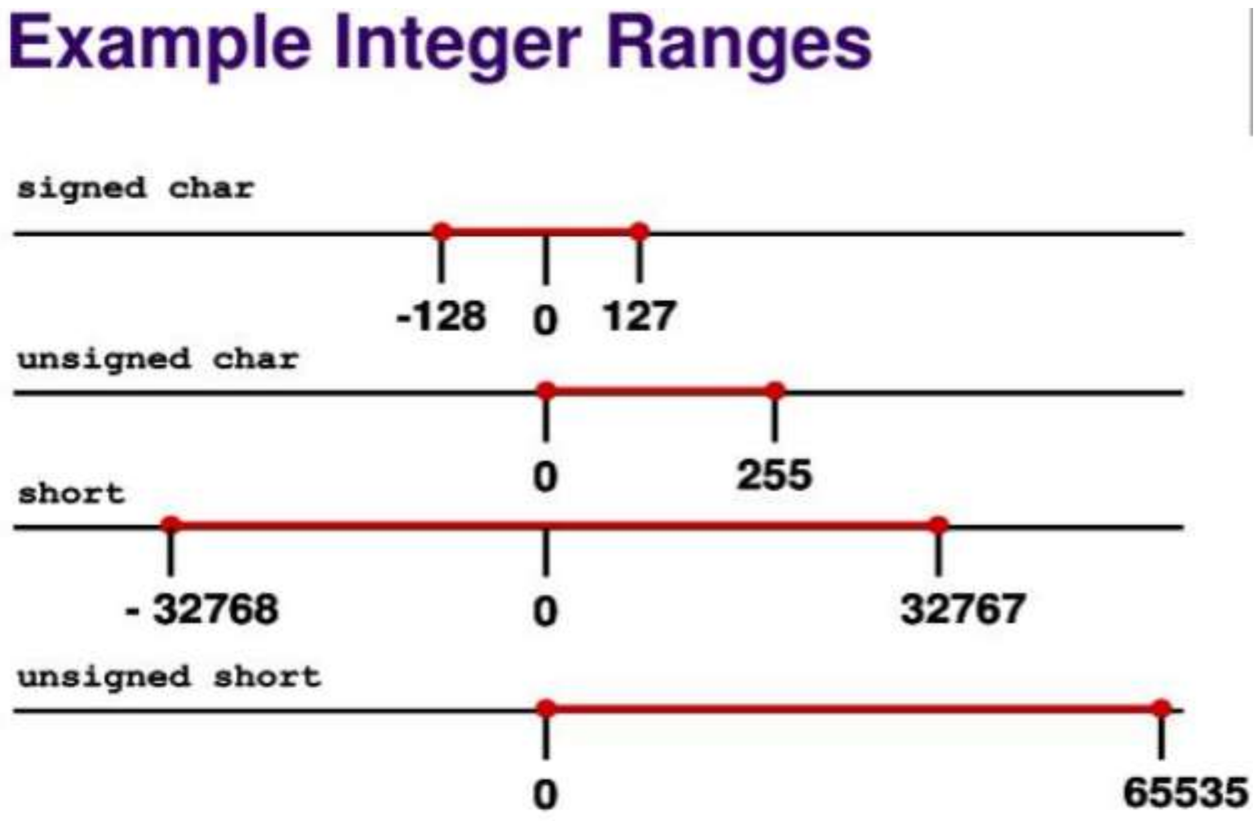
Integer Security

Integer Ranges



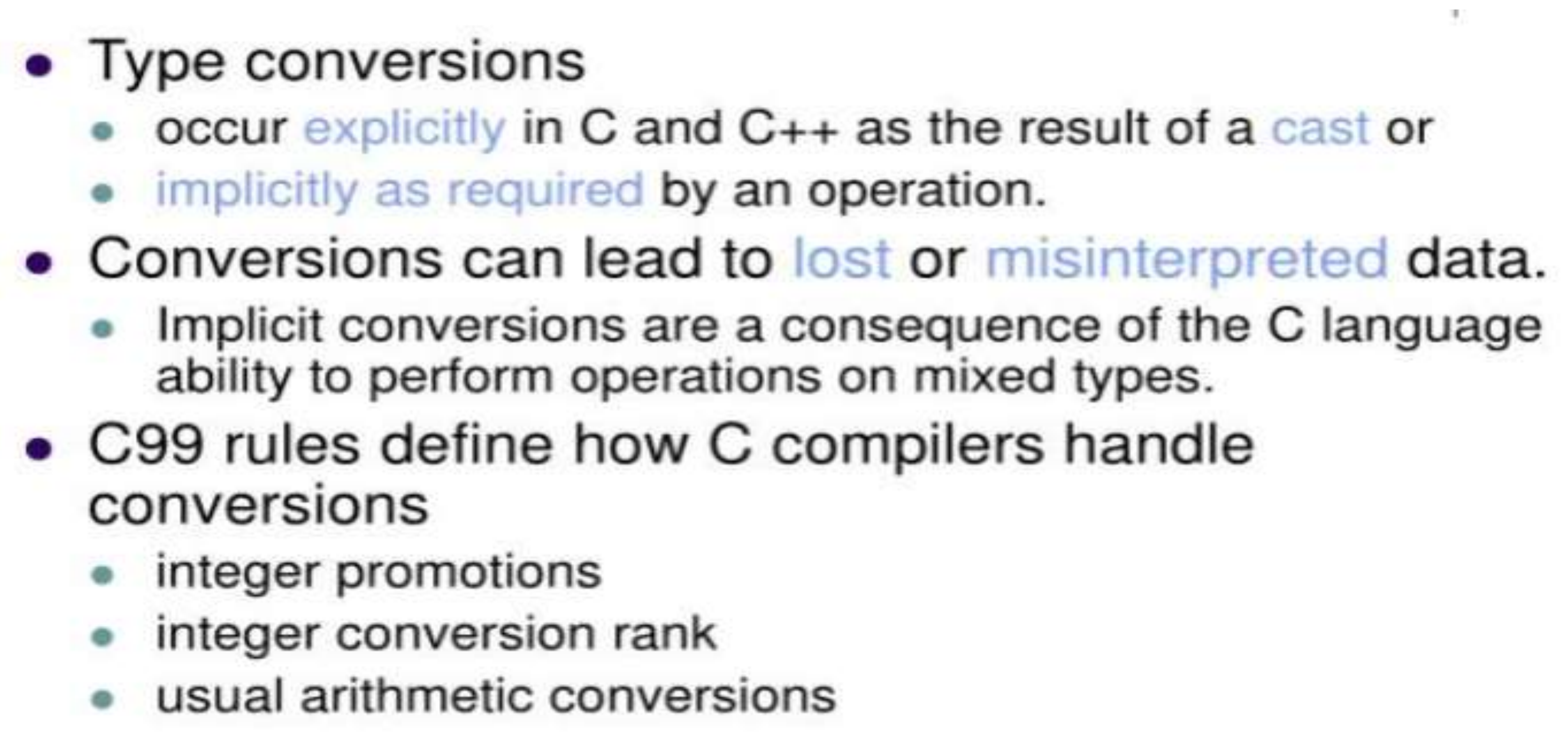
- Minimum and maximum values for an integer type depend on
  - the type’s representation
  - signedness
  - number of allocated bits
- The C99 standard sets minimum requirements for these ranges.

Integer Security



Integer Security

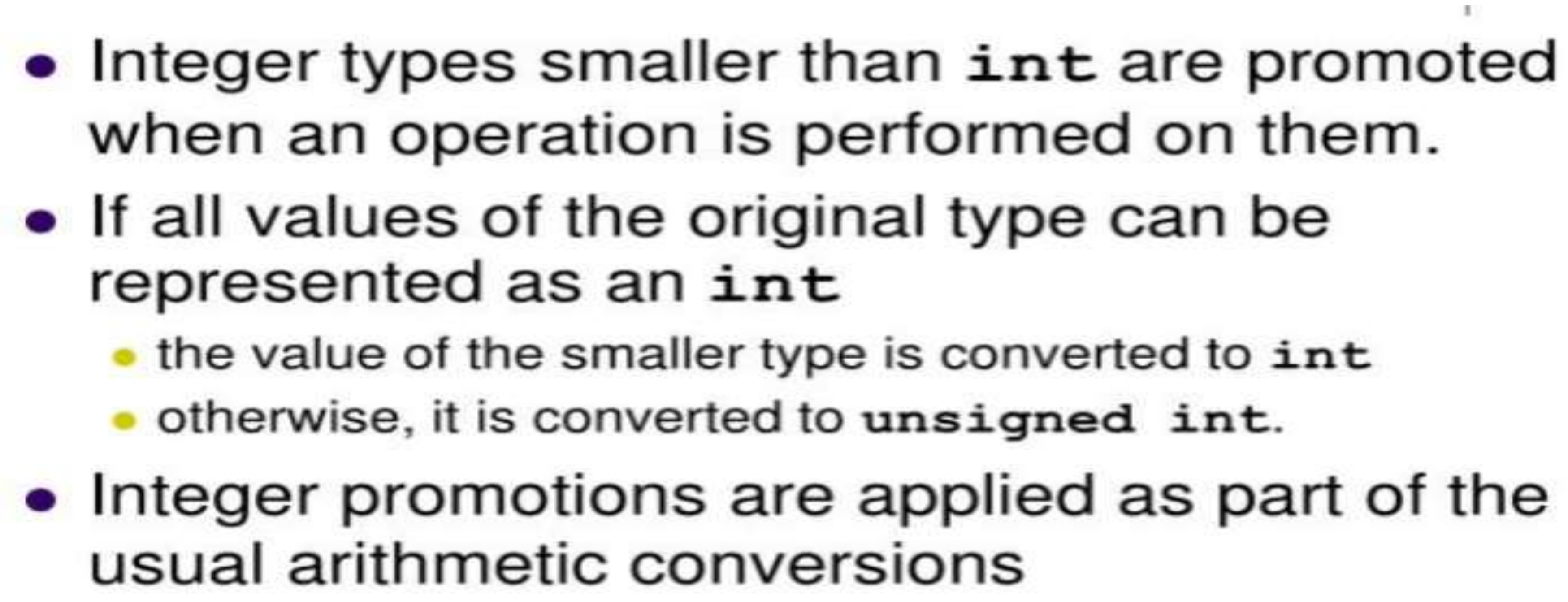
Integer Conversions



- Type conversions
  - occur explicitly in C and C++ as the result of a cast or
  - implicitly as required by an operation.
- Conversions can lead to lost or misinterpreted data.
  - Implicit conversions are a consequence of the C language ability to perform operations on mixed types.
- C99 rules define how C compilers handle conversions
  - integer promotions
  - integer conversion rank
  - usual arithmetic conversions

Integer Security

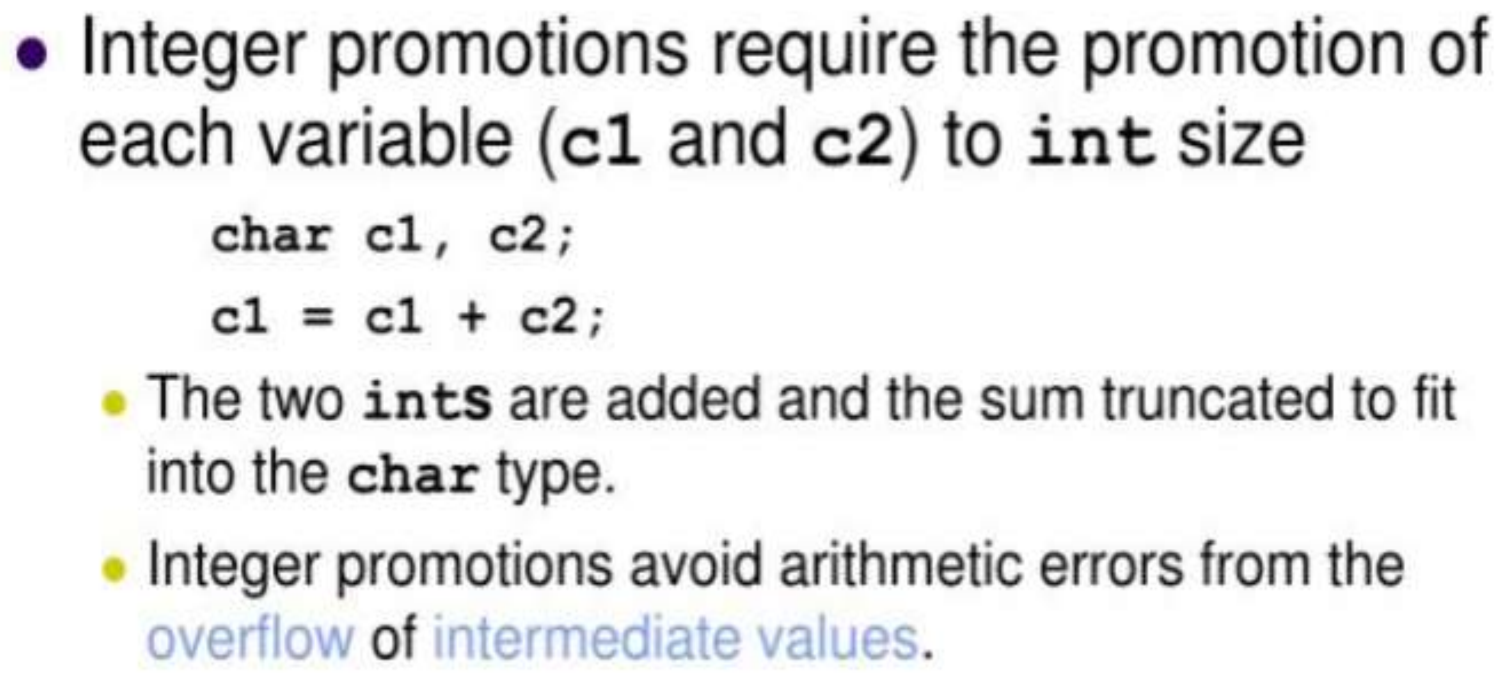
Integer Promotions



- Integer types smaller than **`int`** are promoted when an operation is performed on them.
- If all values of the original type can be represented as an **`int`**
  - the value of the smaller type is converted to **`int`**
  - otherwise, it is converted to **`unsigned int`**.
- Integer promotions are applied as part of the usual arithmetic conversions

Integer Security

Integer Promotions Example



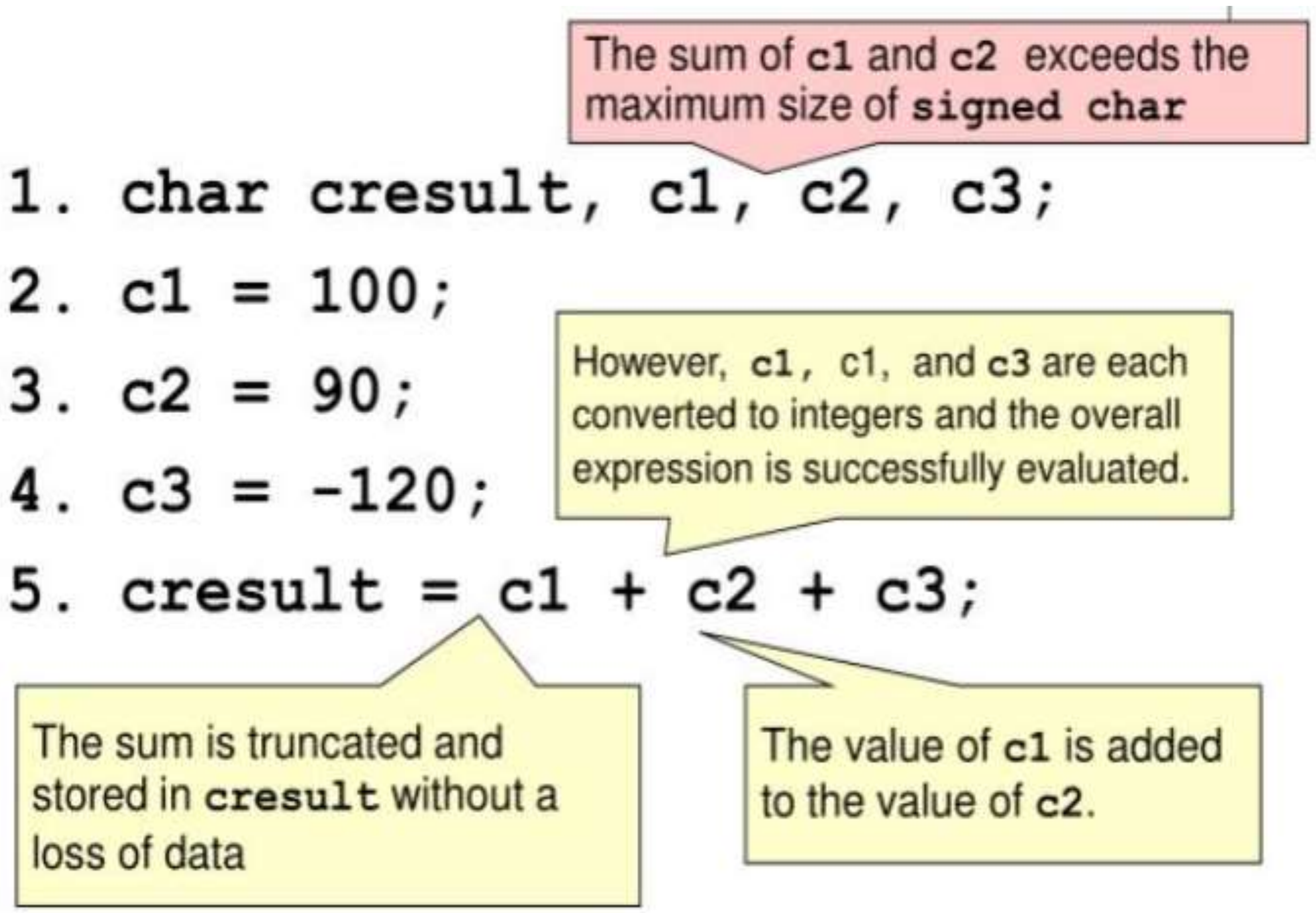
- Integer promotions require the promotion of each variable (**`c1`** and **`c2`**) to **`int`** size

```
char c1, c2;
c1 = c1 + c2;
```

  - The two **`ints`** are added and the sum truncated to fit into the **`char`** type.
  - Integer promotions avoid arithmetic errors from the overflow of intermediate values.

Integer Security

Implicit Conversions



The sum of **c1** and **c2** exceeds the maximum size of **signed char**

```
char cresult, c1, c2, c3;
c1 = 100;
c2 = 90;
c3 = -120;
cresult = c1 + c2 + c3;
```

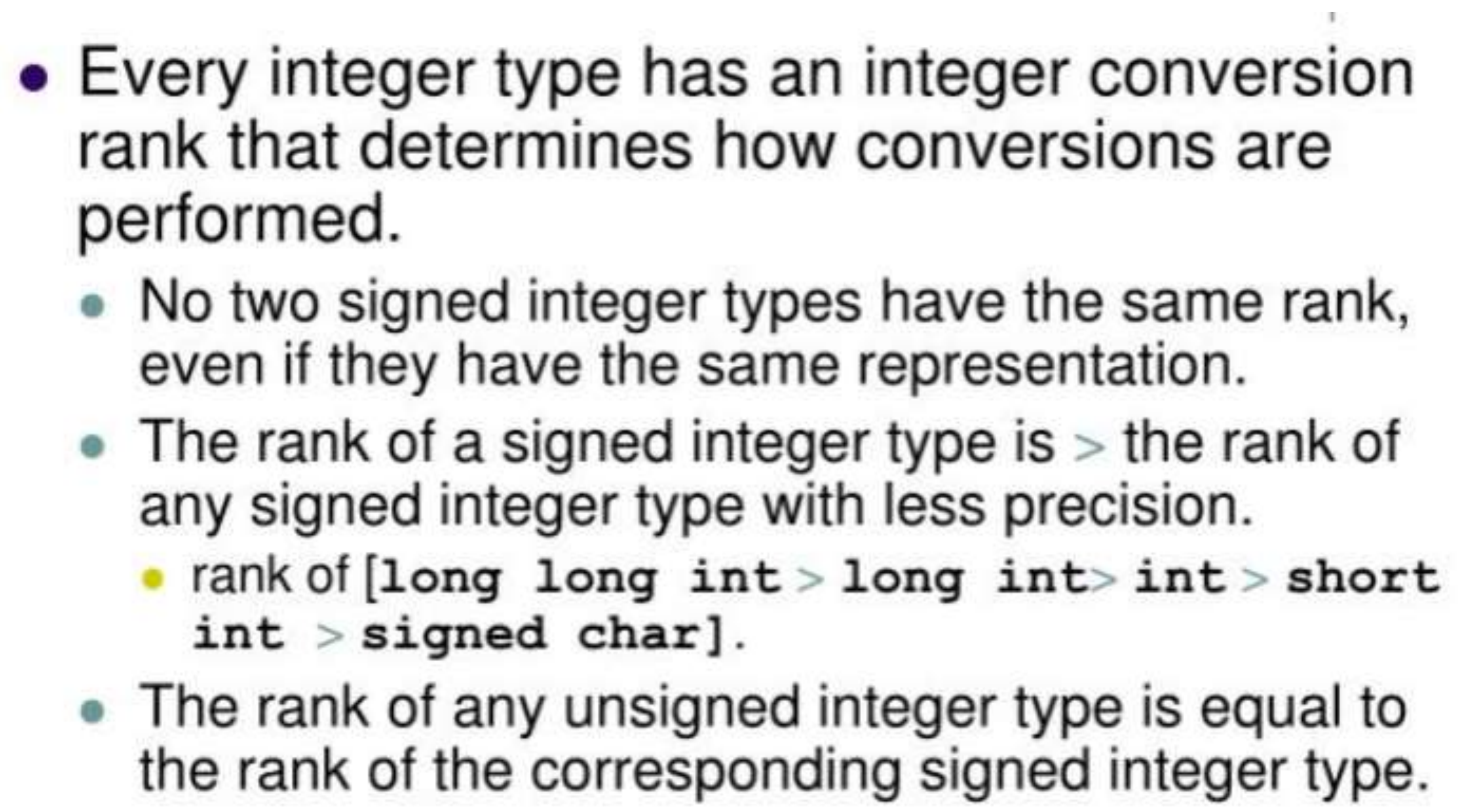
However, **c1**, c1, and **c3** are each converted to integers and the overall expression is successfully evaluated.

The sum is truncated and stored in **cresult** without a loss of data

The value of **c1** is added to the value of **c2**.

Integer Security

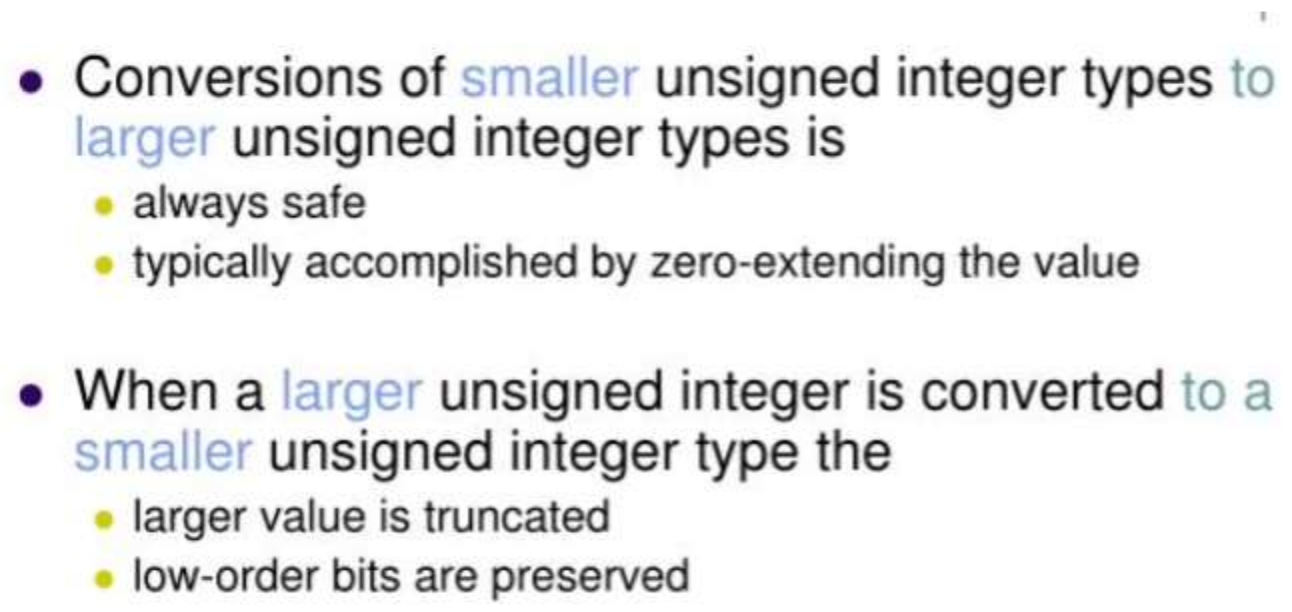
Integer Conversion Ranks and Rules:



- Every integer type has an integer conversion rank that determines how conversions are performed.
  - No two signed integer types have the same rank, even if they have the same representation.
  - The rank of a signed integer type is > the rank of any signed integer type with less precision.
    - rank of [`long long int` > `long int` > `int` > `short int` > `signed char`].
  - The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type.

Integer Security

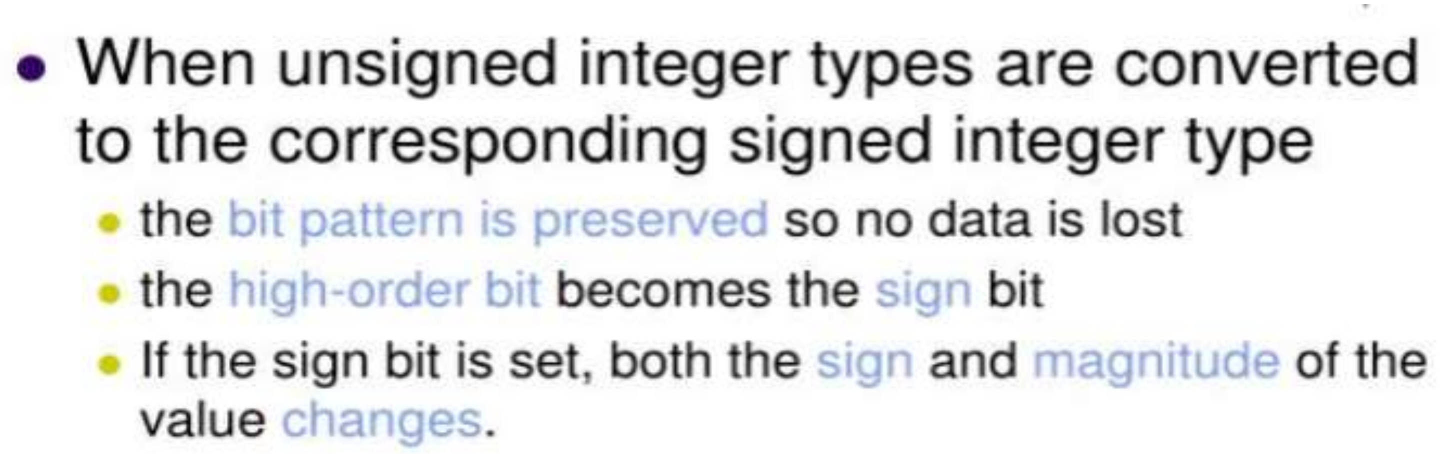
Unsigned Integer Conversions:



- Conversions of smaller unsigned integer types to larger unsigned integer types is
  - always safe
  - typically accomplished by zero-extending the value
- When a larger unsigned integer is converted to a smaller unsigned integer type the
  - larger value is truncated
  - low-order bits are preserved

Integer Security

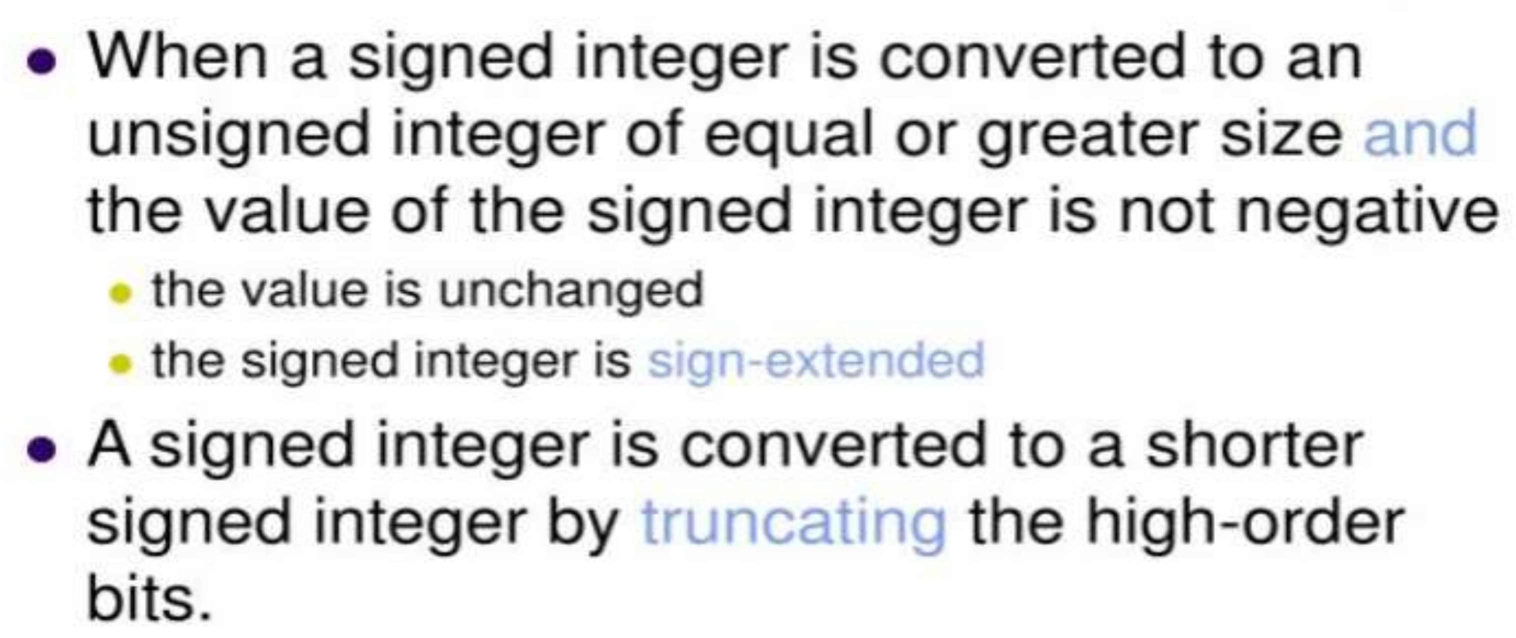
Unsigned Integer Conversions:



- When unsigned integer types are converted to the corresponding signed integer type
  - the bit pattern is preserved so no data is lost
  - the high-order bit becomes the sign bit
  - If the sign bit is set, both the sign and magnitude of the value changes.

Integer Security

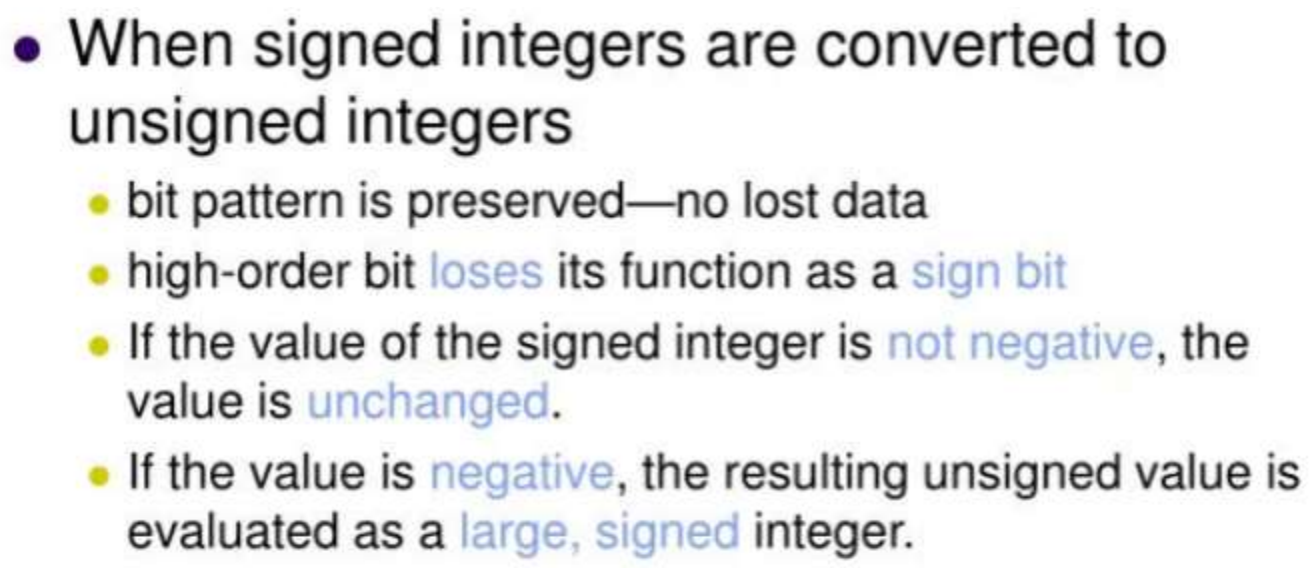
Signed Integer Conversions:



- When a signed integer is converted to an unsigned integer of equal or greater size and the value of the signed integer is not negative
  - the value is unchanged
  - the signed integer is sign-extended
- A signed integer is converted to a shorter signed integer by truncating the high-order bits.

Integer Security

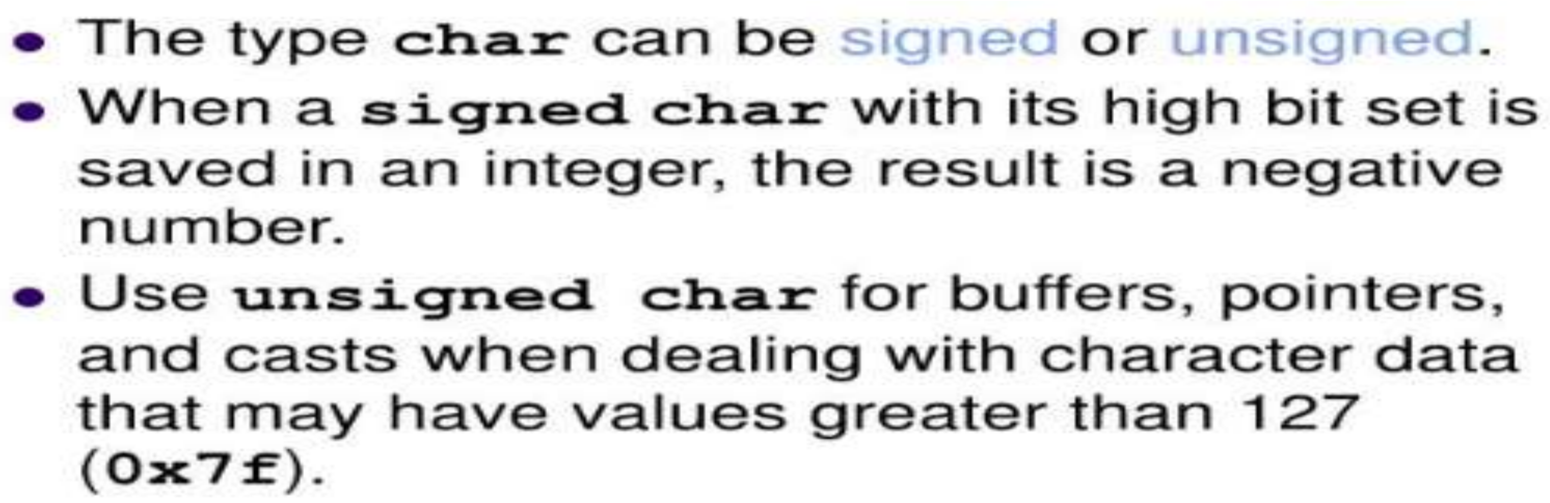
Signed Integer Conversions:



- When signed integers are converted to unsigned integers
  - bit pattern is preserved—no lost data
  - high-order bit loses its function as a sign bit
  - If the value of the signed integer is not negative, the value is unchanged.
  - If the value is negative, the resulting unsigned value is evaluated as a large, signed integer.

Integer Security

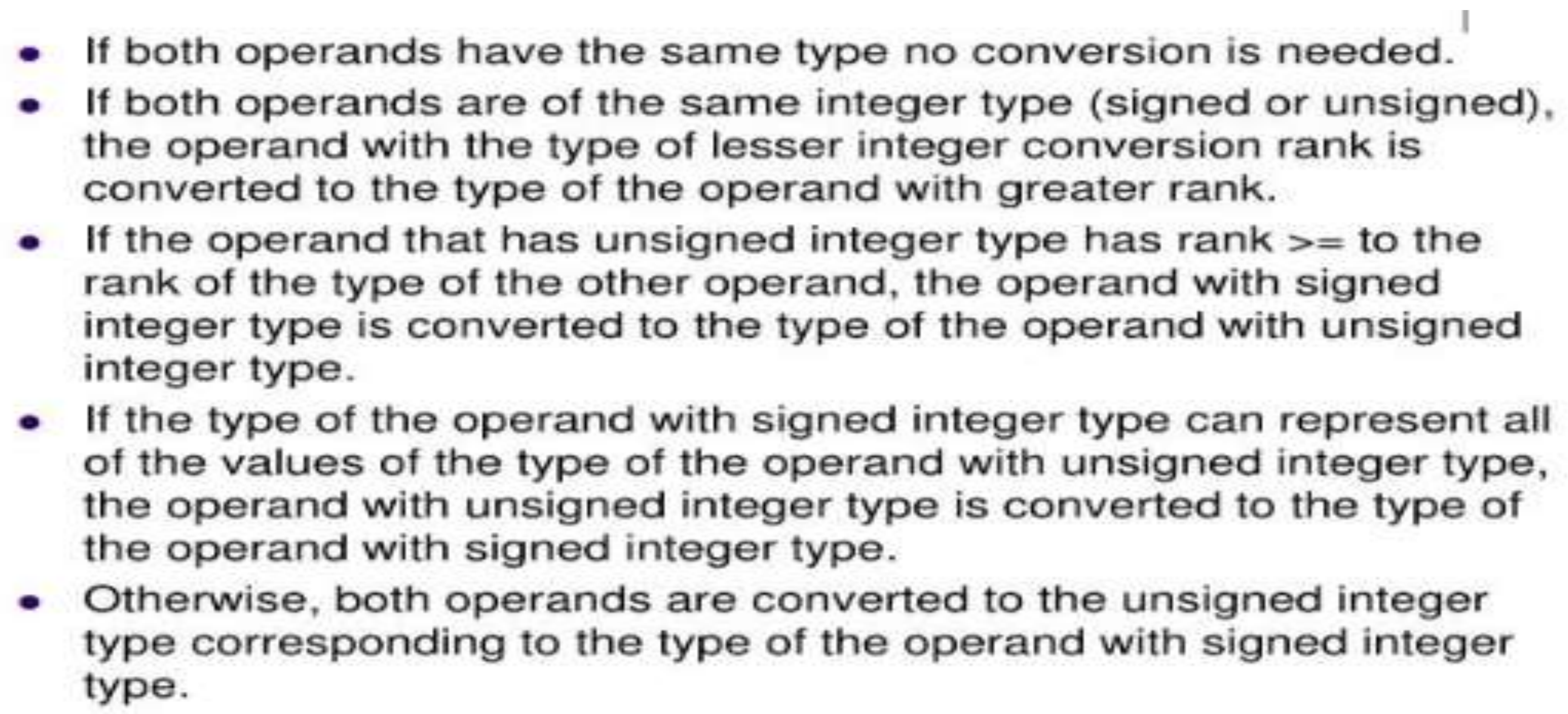
Signed/Unsigned Characters:



- The type **`char`** can be signed or unsigned.
- When a **`signed char`** with its high bit set is saved in an integer, the result is a negative number.
- Use **`unsigned char`** for buffers, pointers, and casts when dealing with character data that may have values greater than 127 (**`0x7f`**).

Integer Security

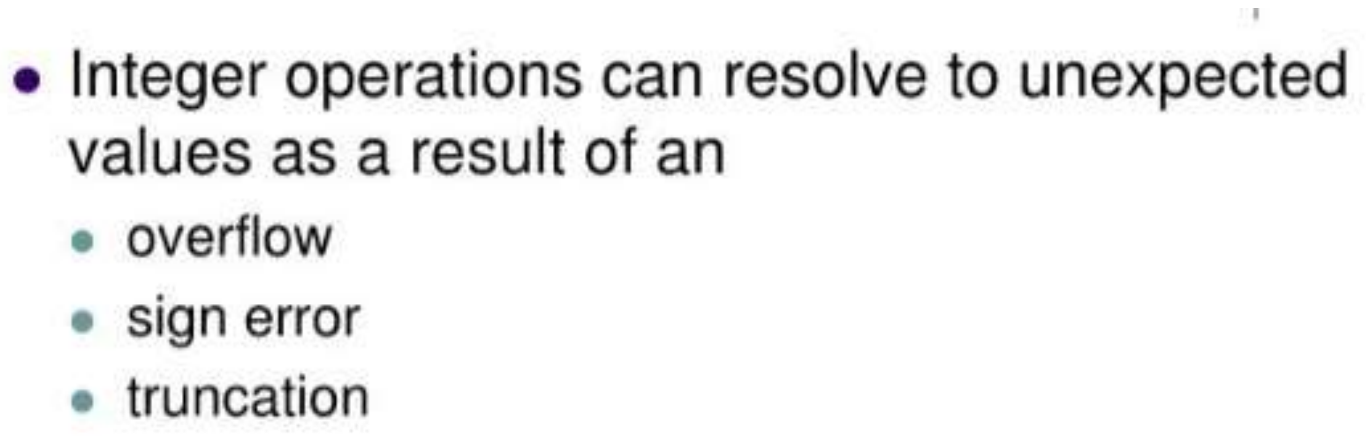
Usual Arithmetic Conversions:



- If both operands have the same type no conversion is needed.
- If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
- If the operand that has unsigned integer type has rank >= to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
- If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
- Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Integer Security

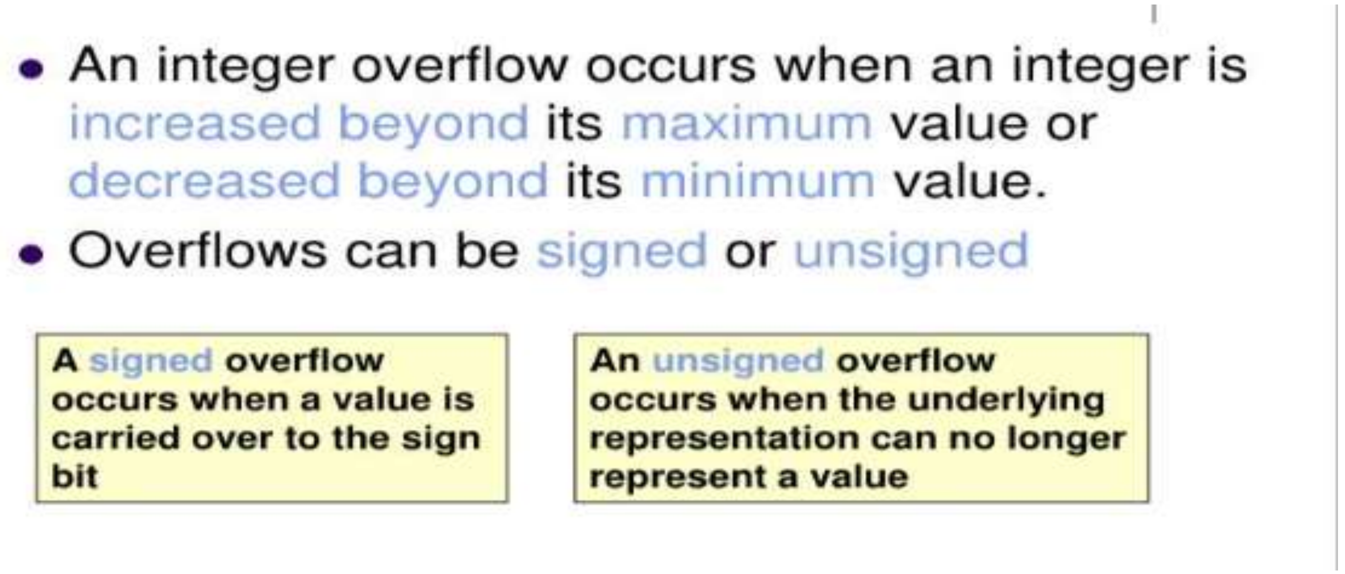
Integer Error Conditions



- Integer operations can resolve to unexpected values as a result of an
  - overflow
  - sign error
  - truncation

Integer Security

Overflow



- An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.
- Overflows can be signed or unsigned

A signed overflow occurs when a value is carried over to the sign bit

An unsigned overflow occurs when the underlying representation can no longer represent a value

Integer Security

Signed Overflow Example:

```
#include <stdio.h>
#include <limits.h>  // For INT_MAX

int main() {
    int num = INT_MAX;  // INT_MAX is the maximum value for an int
    printf("Before overflow: %d\n", num);

    num = num + 1;  // This causes an overflow

    printf("After overflow: %d\n", num);  // The result will wrap around to the minimum value
    return 0;
}
```

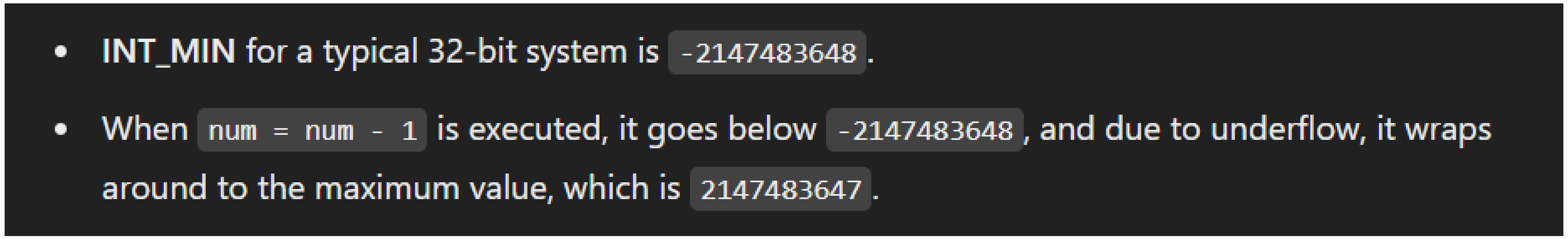
Output:-

```
Before overflow: 2147483647
After overflow: -2147483648
```

Integer Security

Signed underflow Example:

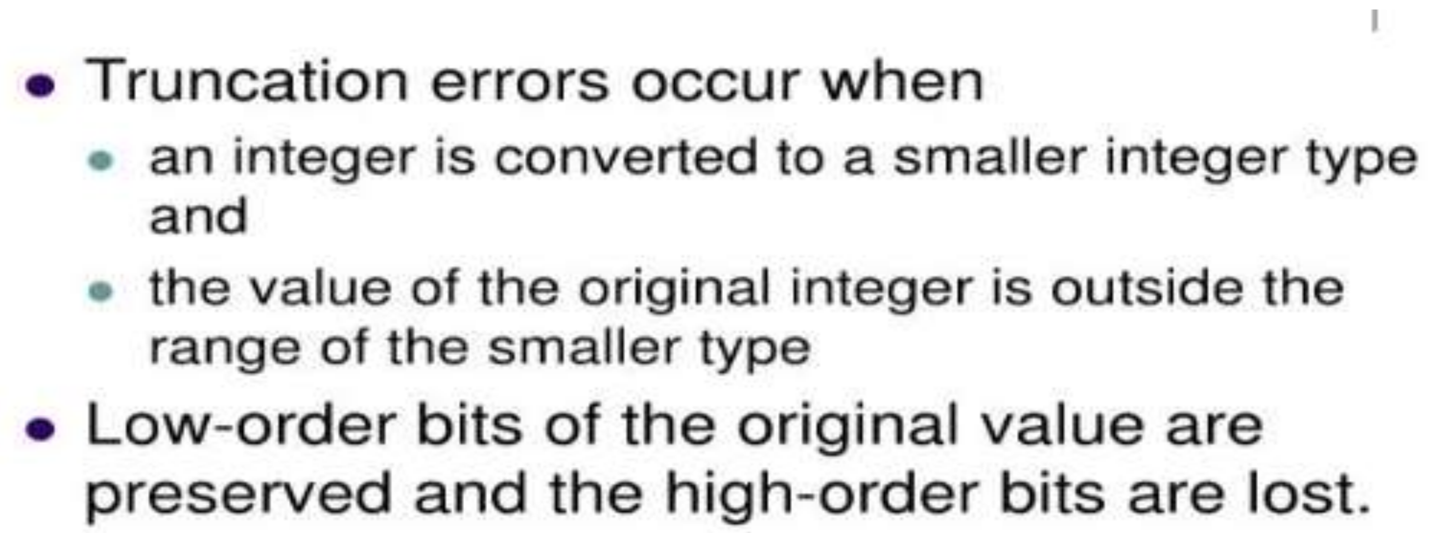
```
#include <stdio.h>
#include <limits.h>  // For INT_MIN

int main() {
    int num = INT_MIN;  // INT_MIN is the minimum value for an int
    printf("Before underflow: %d\n", num);

    num = num - 1;  // This causes an underflow

    printf("After underflow: %d\n", num);  // The result will wrap around to the maximum value
    return 0;
}
```



- INT_MIN for a typical 32-bit system is `-2147483648`.
- When `num = num - 1` is executed, it goes below `-2147483648`, and due to underflow, it wraps around to the maximum value, which is `2147483647`.

Integer Security

Truncation Errors:

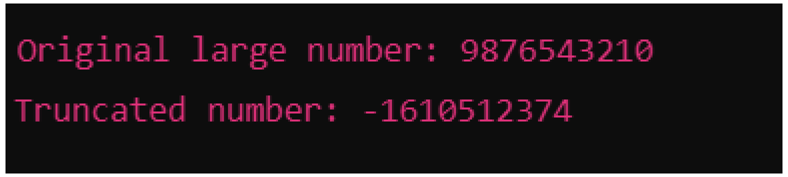


- Truncation errors occur when
  - an integer is converted to a smaller integer type and
  - the value of the original integer is outside the range of the smaller type
- Low-order bits of the original value are preserved and the high-order bits are lost.

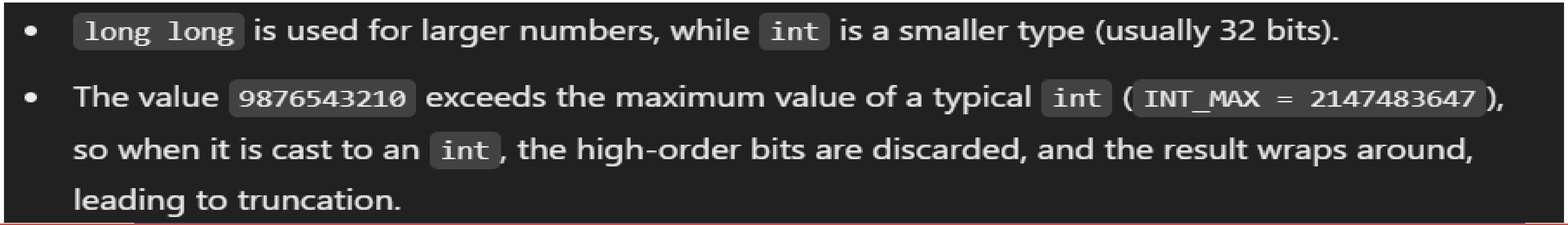
Integer Security

Truncation Errors:

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

int main() {
    long long largeNumber = 9876543210;  // Large number (larger than INT_MAX)
    int smallerNumber;

    smallerNumber = (int) largeNumber;  // Truncation from long long to int

    printf("Original large number: %lld\n", largeNumber);
    printf("Truncated number: %d\n", smallerNumber);

    return 0;
}
```



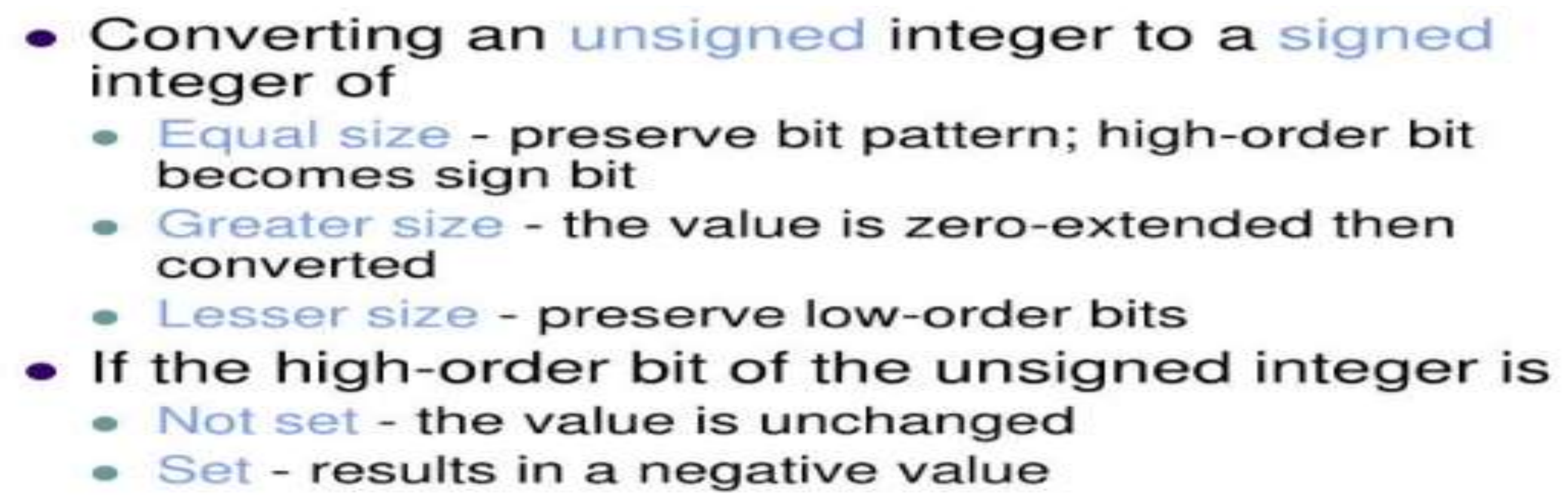
```
Original large number: 9876543210
Truncated number: -1610512374
```



- `long long` is used for larger numbers, while `int` is a smaller type (usually 32 bits).
- The value `9876543210` exceeds the maximum value of a typical `int` (`INT_MAX = 2147483647`), so when it is cast to an `int`, the high-order bits are discarded, and the result wraps around, leading to truncation.

Integer Security

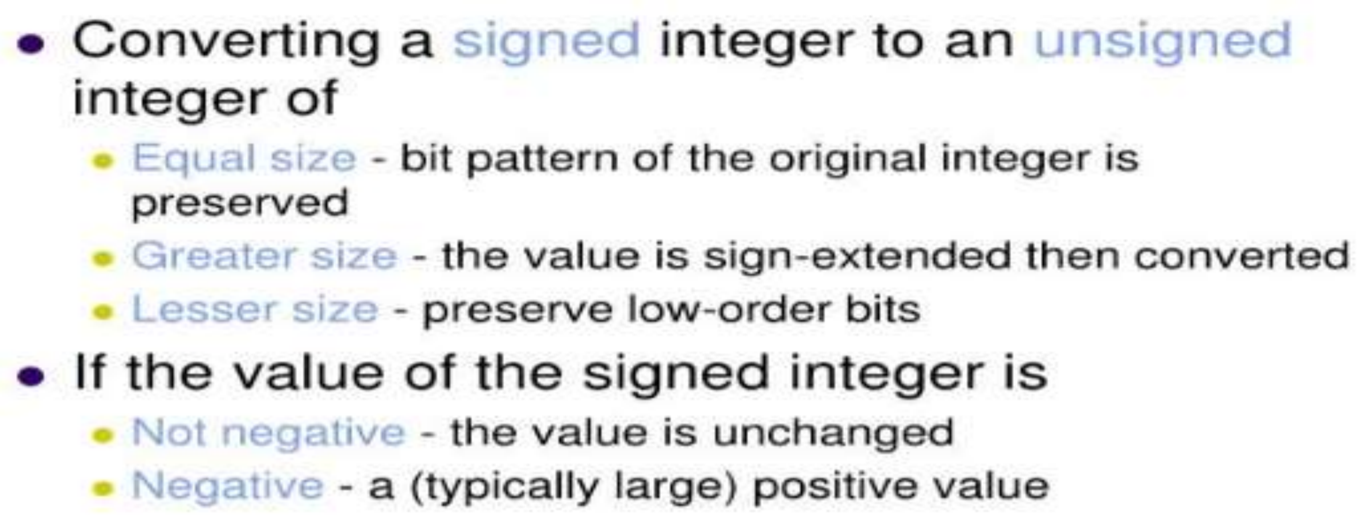
Signed Errors:



- Converting an unsigned integer to a signed integer of
  - Equal size - preserve bit pattern; high-order bit becomes sign bit
  - Greater size - the value is zero-extended then converted
  - Lesser size - preserve low-order bits
- If the high-order bit of the unsigned integer is
  - Not set - the value is unchanged
  - Set - results in a negative value

Integer Security

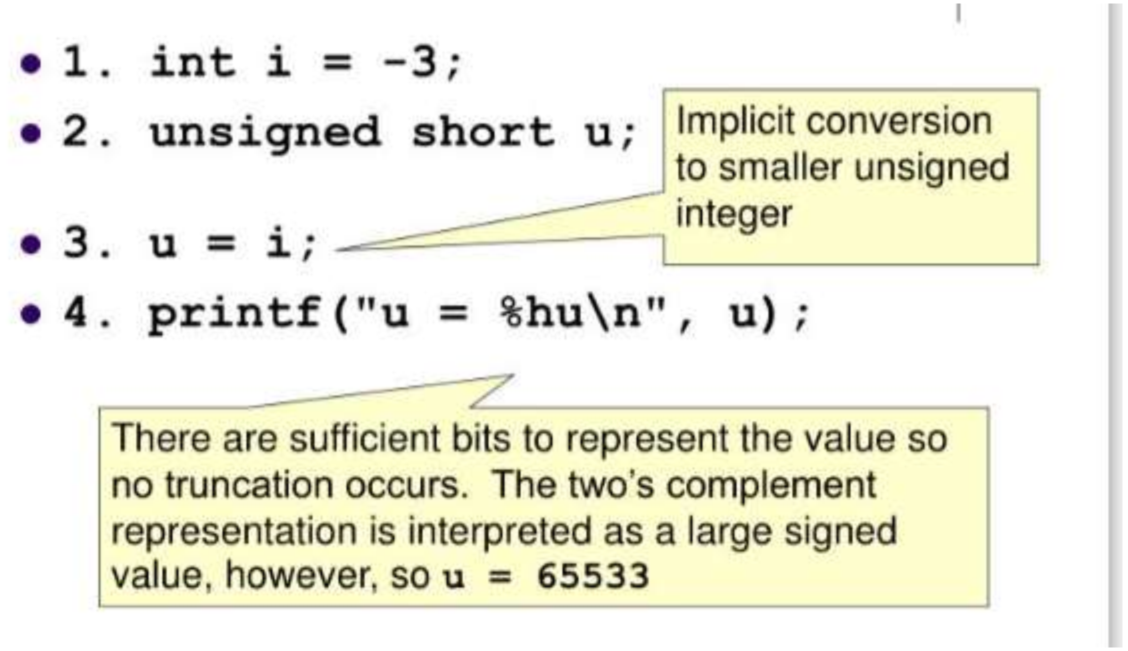
Converting to unsigned integer:



- Converting a signed integer to an unsigned integer of
  - Equal size - bit pattern of the original integer is preserved
  - Greater size - the value is sign-extended then converted
  - Lesser size - preserve low-order bits
- If the value of the signed integer is
  - Not negative - the value is unchanged
  - Negative - a (typically large) positive value

Integer Security

Sign Errors:



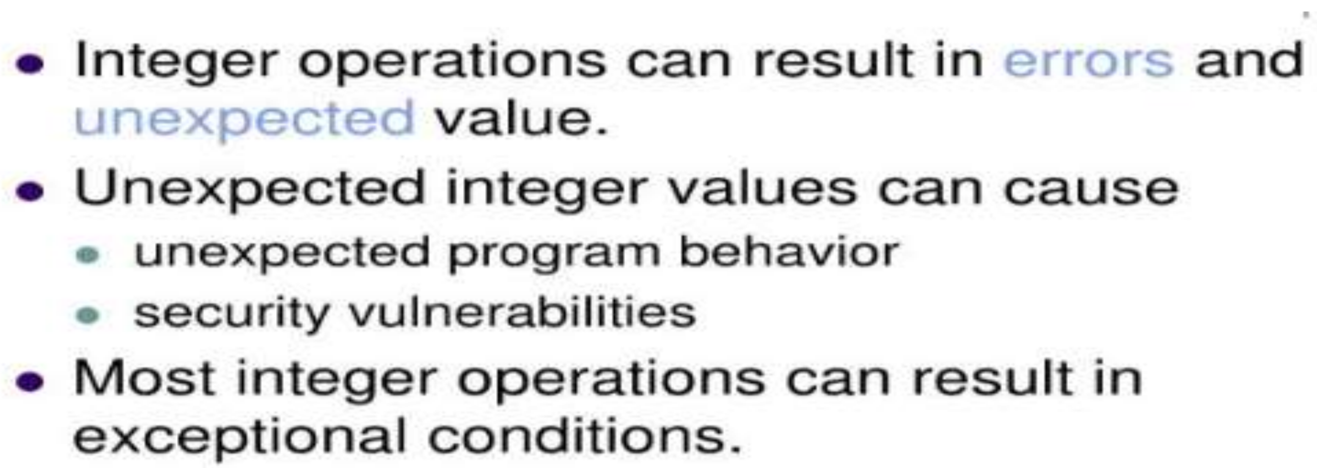
```
int i = -3;
unsigned short u;
u = i;
printf("u = %hu\n", u);
```

Implicit conversion to smaller unsigned integer

There are sufficient bits to represent the value so no truncation occurs. The two's complement representation is interpreted as a large signed value, however, so `u = 65533`

Integer Security

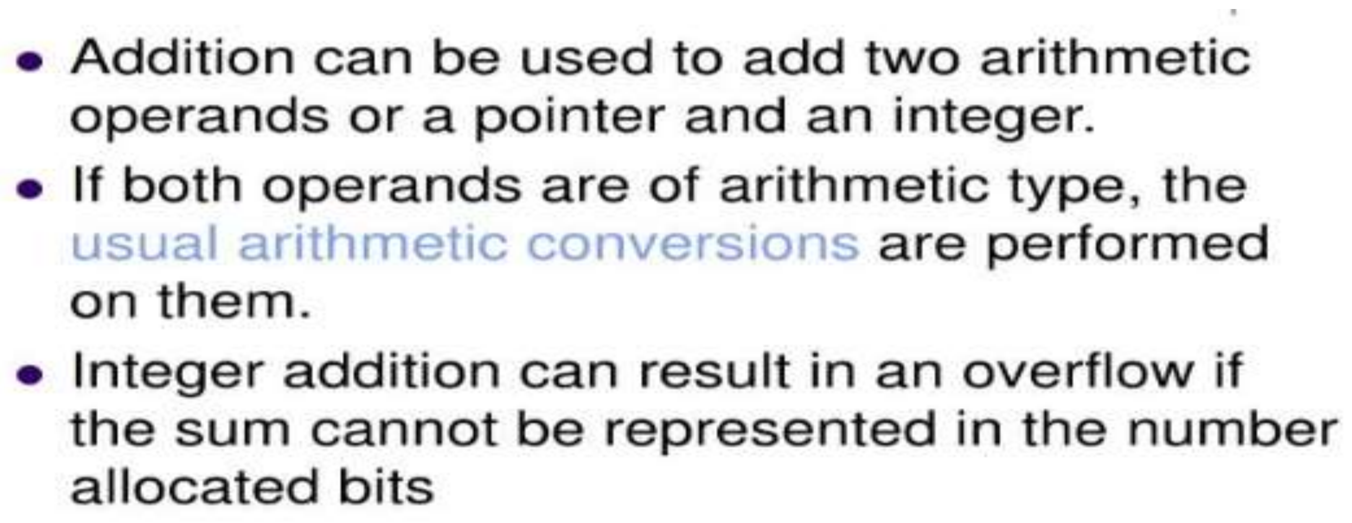
Integer Operations:



- Integer operations can result in errors and unexpected value.
- Unexpected integer values can cause
  - unexpected program behavior
  - security vulnerabilities
- Most integer operations can result in exceptional conditions.

Integer Security

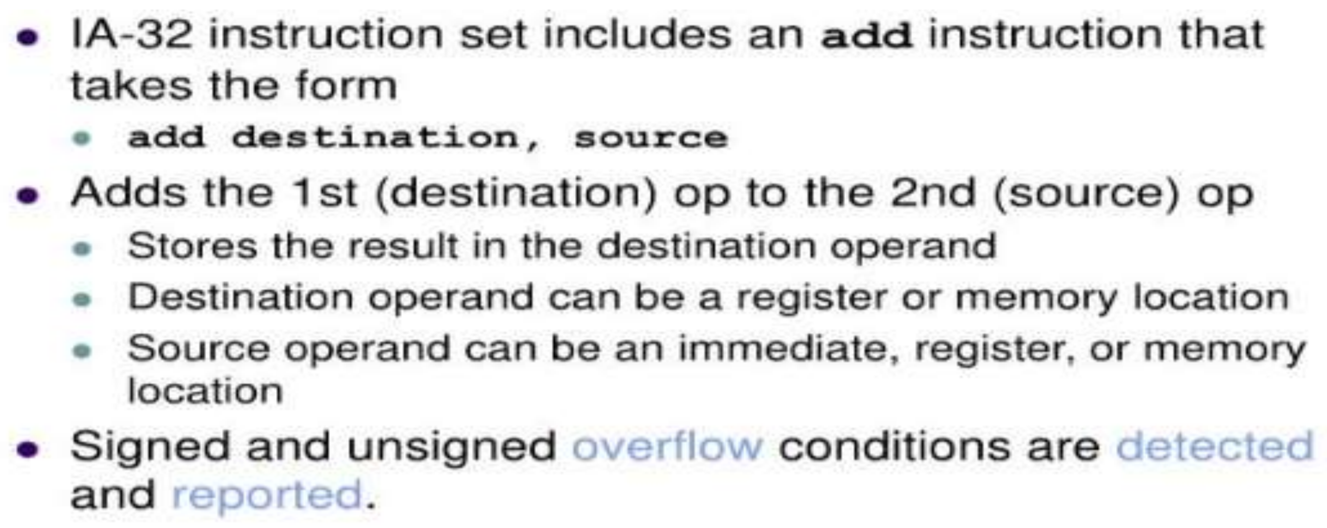
Integer Addition:



- Addition can be used to add two arithmetic operands or a pointer and an integer.
- If both operands are of arithmetic type, the usual arithmetic conversions are performed on them.
- Integer addition can result in an overflow if the sum cannot be represented in the number allocated bits

Integer Security

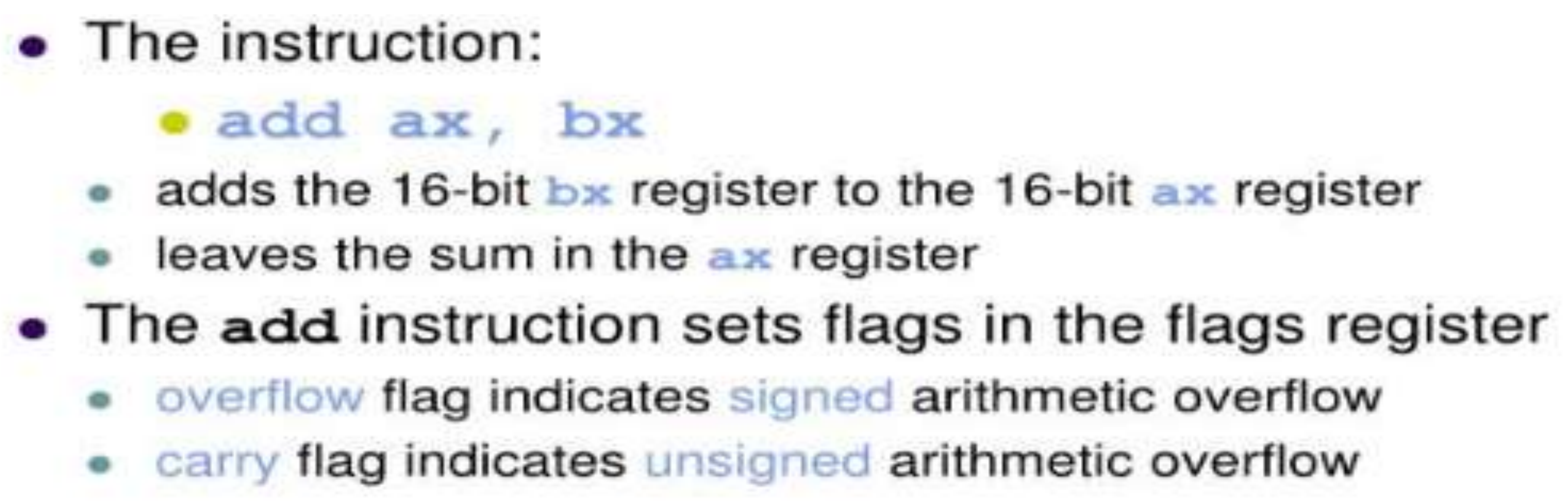
Add Instructions:



- IA-32 instruction set includes an **`add`** instruction that takes the form
  - **`add destination, source`**
- Adds the 1st (destination) op to the 2nd (source) op
  - Stores the result in the destination operand
  - Destination operand can be a register or memory location
  - Source operand can be an immediate, register, or memory location
- Signed and unsigned overflow conditions are detected and reported.

Integer Security

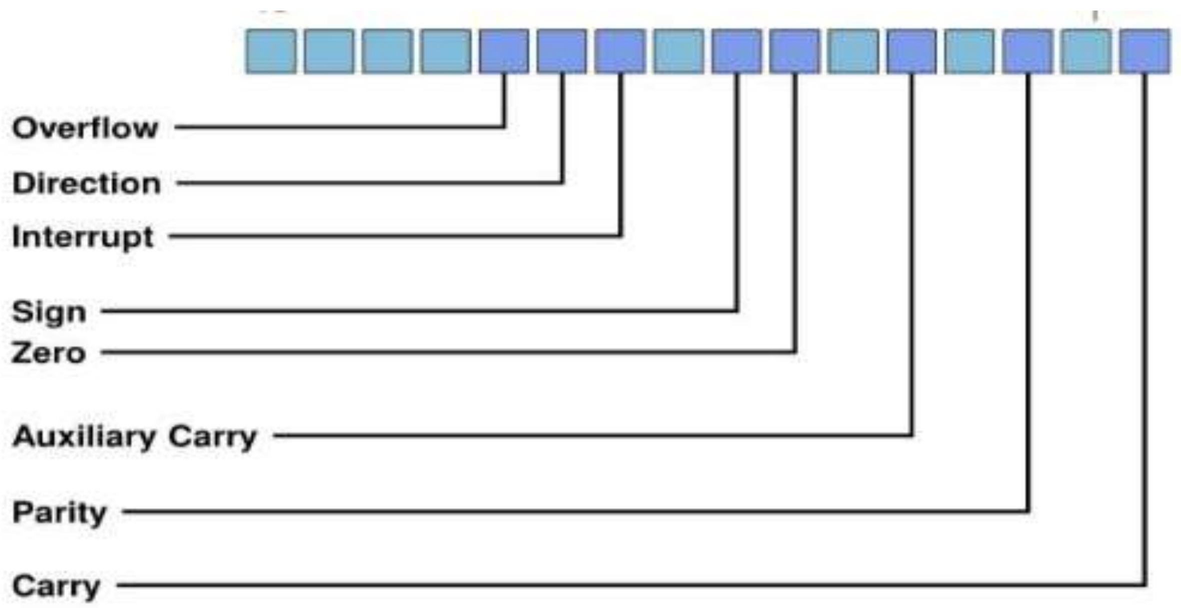
Add Instructions Example:



- The instruction:
  - `add ax, bx`
  - adds the 16-bit `bx` register to the 16-bit `ax` register
  - leaves the sum in the `ax` register
- The **add** instruction sets flags in the flags register
  - overflow flag indicates signed arithmetic overflow
  - carry flag indicates unsigned arithmetic overflow

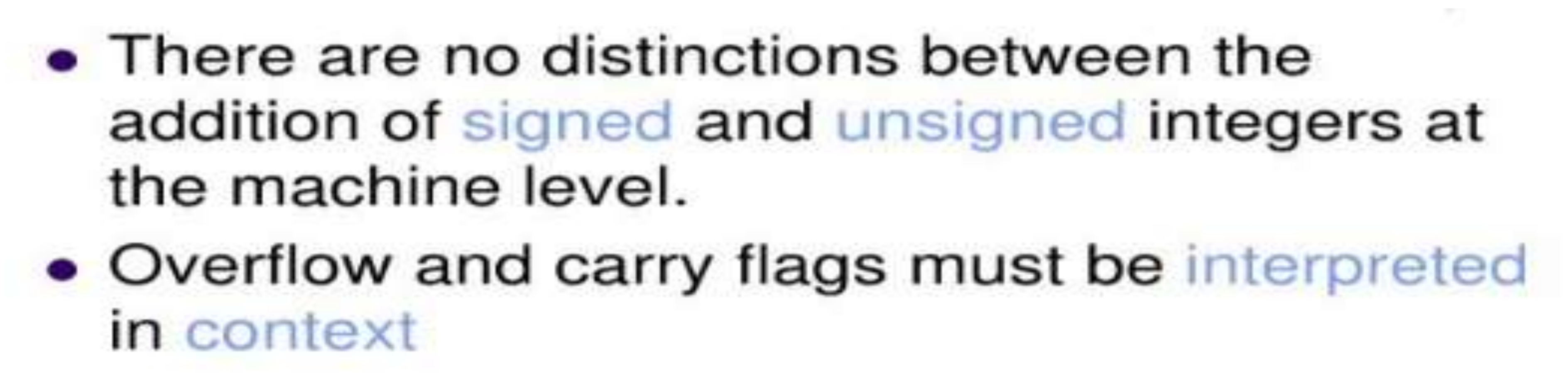
Integer Security

Add Instructions Example:



Integer Security

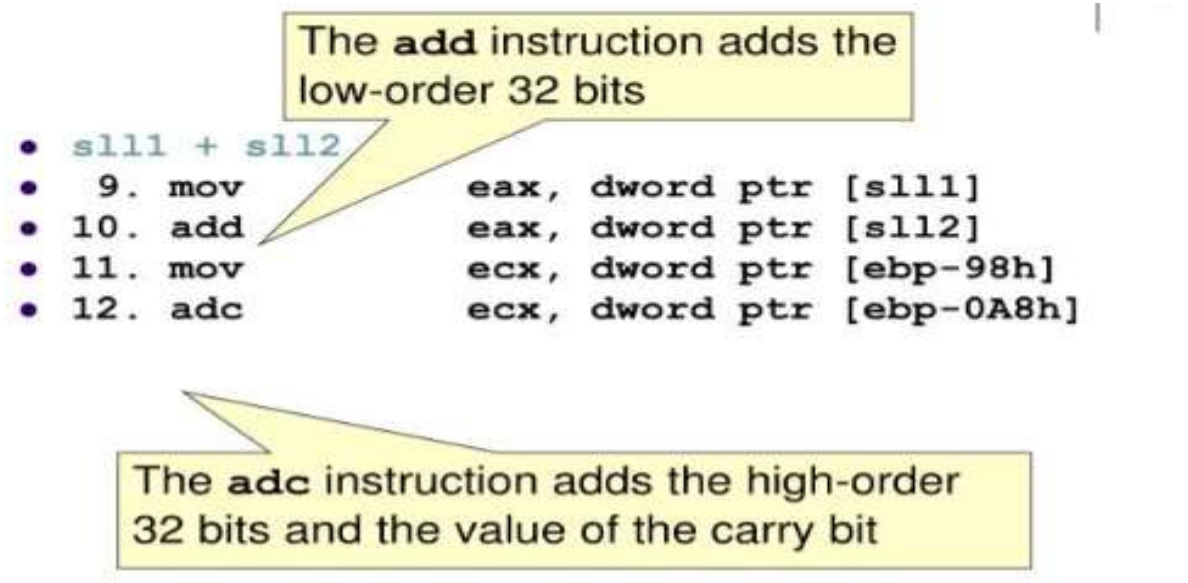
Interpreting Flags:



- There are no distinctions between the addition of signed and unsigned integers at the machine level.
- Overflow and carry flags must be interpreted in context

Integer Security

Added Signed long long int:



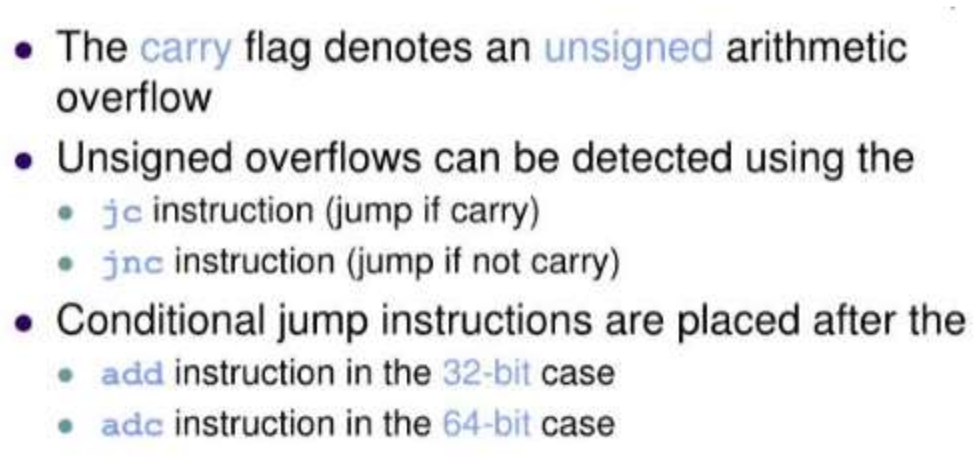
The **add** instruction adds the low-order 32 bits

```
sll1 + sll2
mov           eax, dword ptr [sll1]
add           eax, dword ptr [sll2]
mov           ecx, dword ptr [ebp-98h]
adc           ecx, dword ptr [ebp-0A8h]
```

The **adc** instruction adds the high-order 32 bits and the value of the carry bit

Integer Security

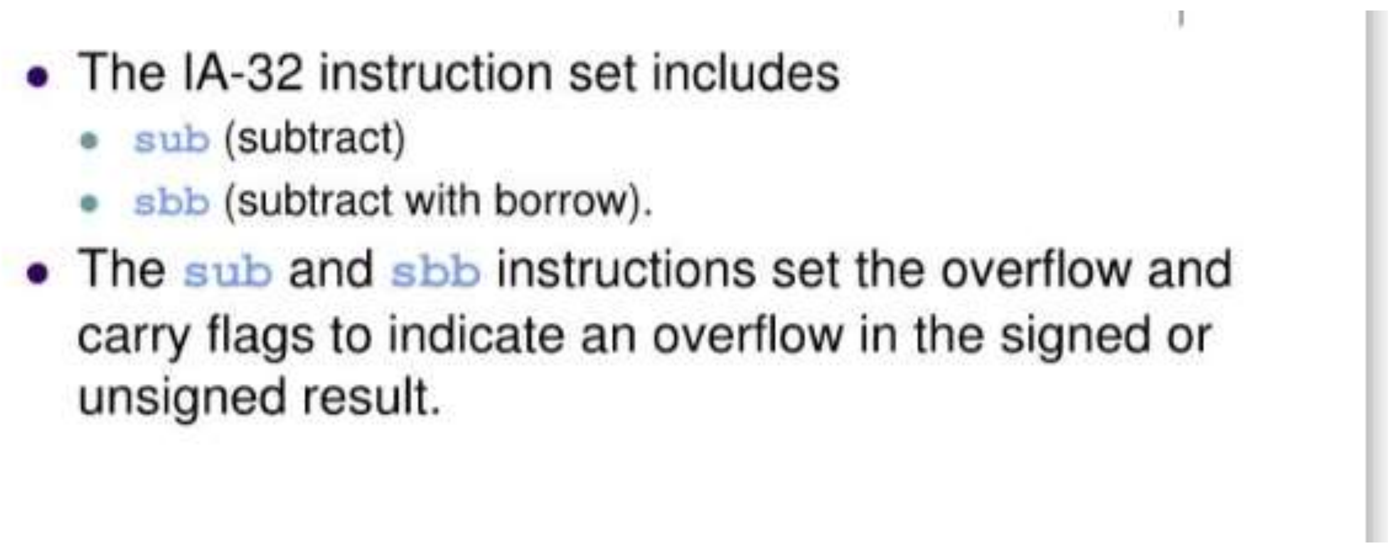
Unsigned overflow detection:



- The carry flag denotes an unsigned arithmetic overflow
- Unsigned overflows can be detected using the
  - `jc` instruction (jump if carry)
  - `jnc` instruction (jump if not carry)
- Conditional jump instructions are placed after the
  - `add` instruction in the 32-bit case
  - `adc` instruction in the 64-bit case

Integer Security

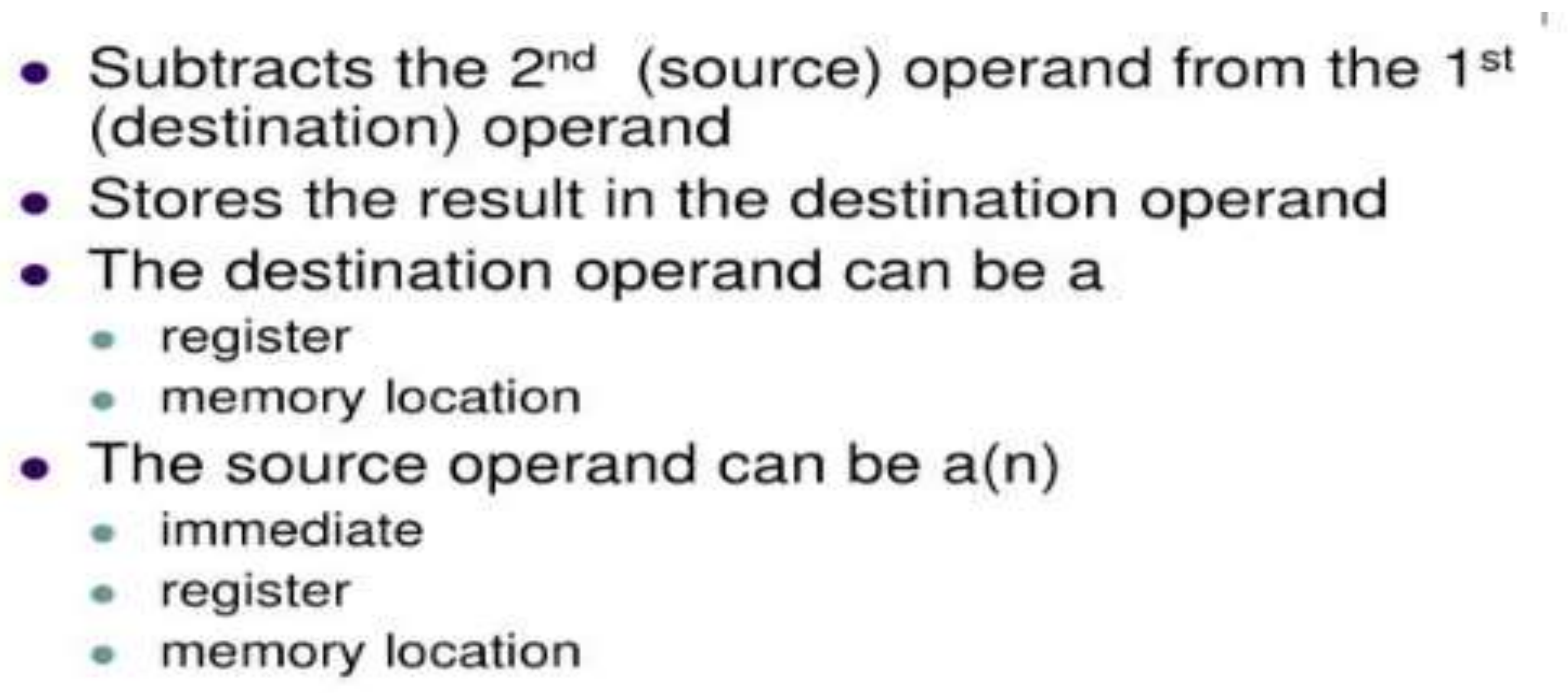
Substraction:



- The IA-32 instruction set includes
  - `sub` (subtract)
  - `sbb` (subtract with borrow).
- The `sub` and `sbb` instructions set the overflow and carry flags to indicate an overflow in the signed or unsigned result.

Integer Security

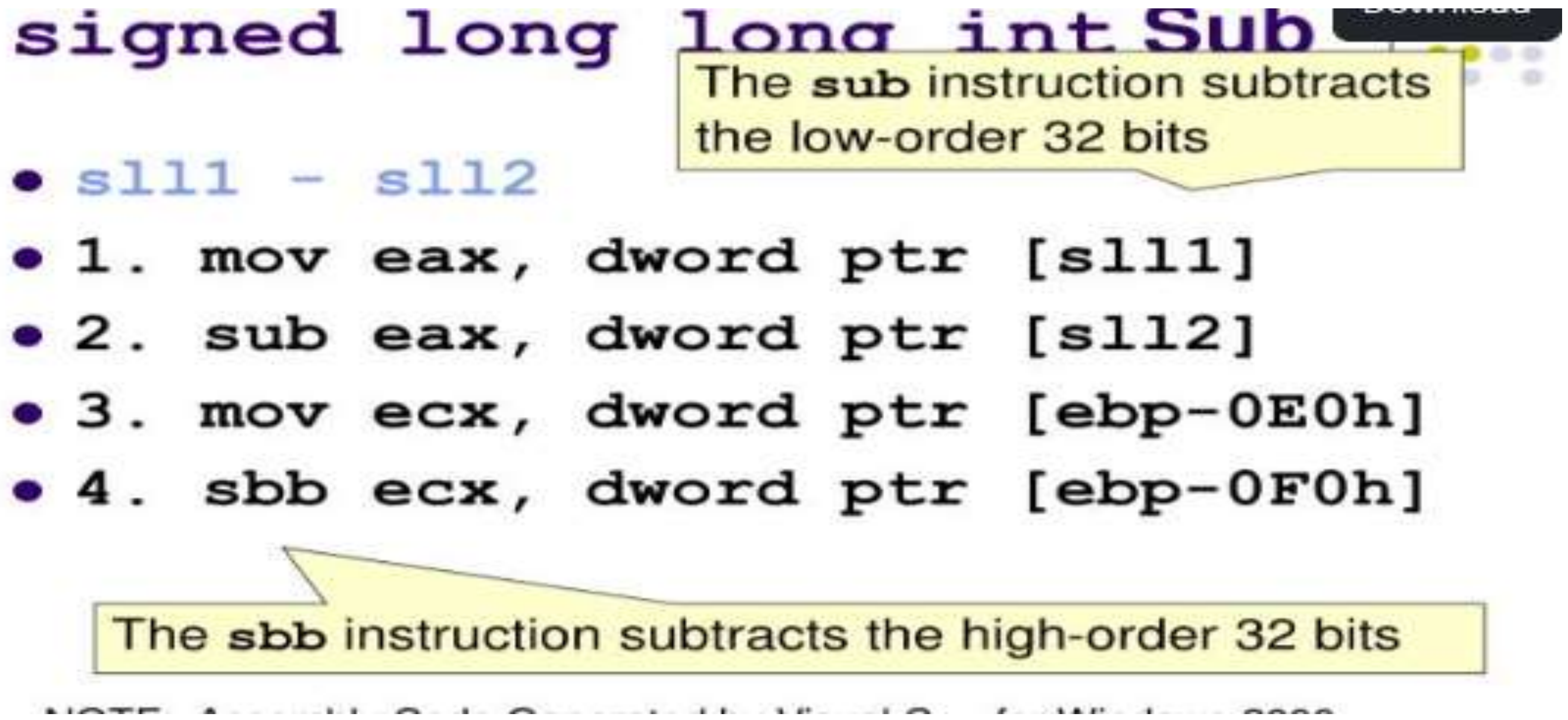
Sub Instruction:-



- Subtracts the 2nd (source) operand from the 1st (destination) operand
- Stores the result in the destination operand
- The destination operand can be a
  - register
  - memory location
- The source operand can be a(n)
  - immediate
  - register
  - memory location

Integer Security

Sub Instruction:-



signed long long int Sub

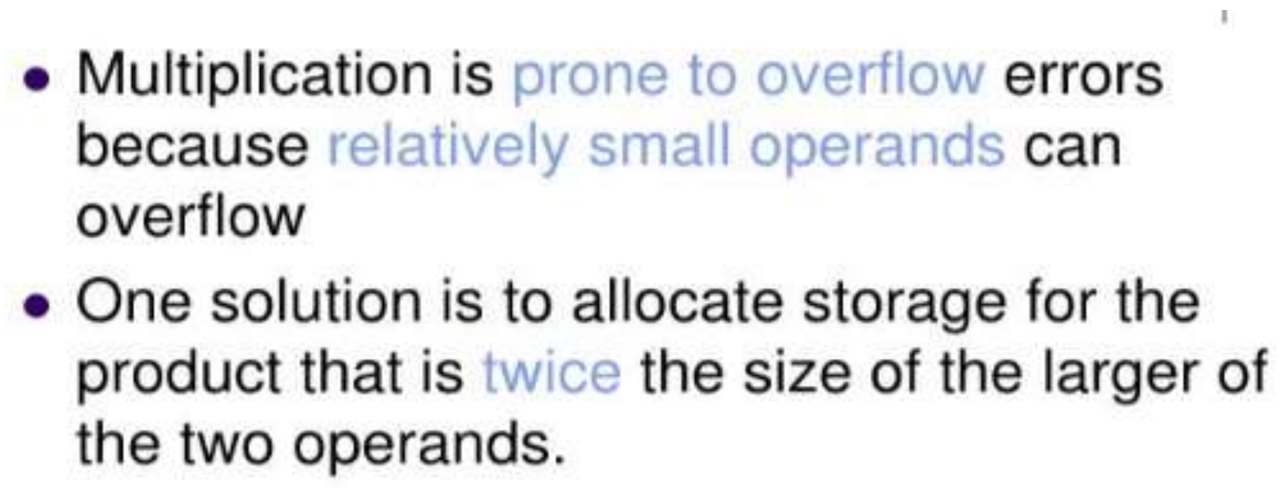
The sub instruction subtracts the low-order 32 bits

- sll1 - sll2
- 1. mov eax, dword ptr [sll1]
- 2. sub eax, dword ptr [sll2]
- 3. mov ecx, dword ptr [ebp-0E0h]
- 4. sbb ecx, dword ptr [ebp-0F0h]

The sbb instruction subtracts the high-order 32 bits

Integer Security

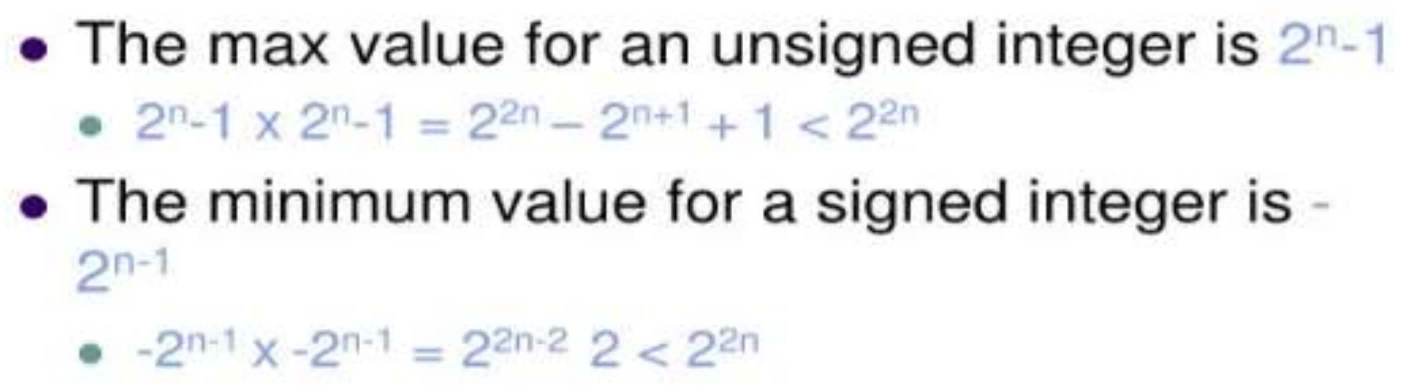
Integer Multiplication:-



- Multiplication is prone to overflow errors because relatively small operands can overflow
- One solution is to allocate storage for the product that is twice the size of the larger of the two operands.

Integer Security

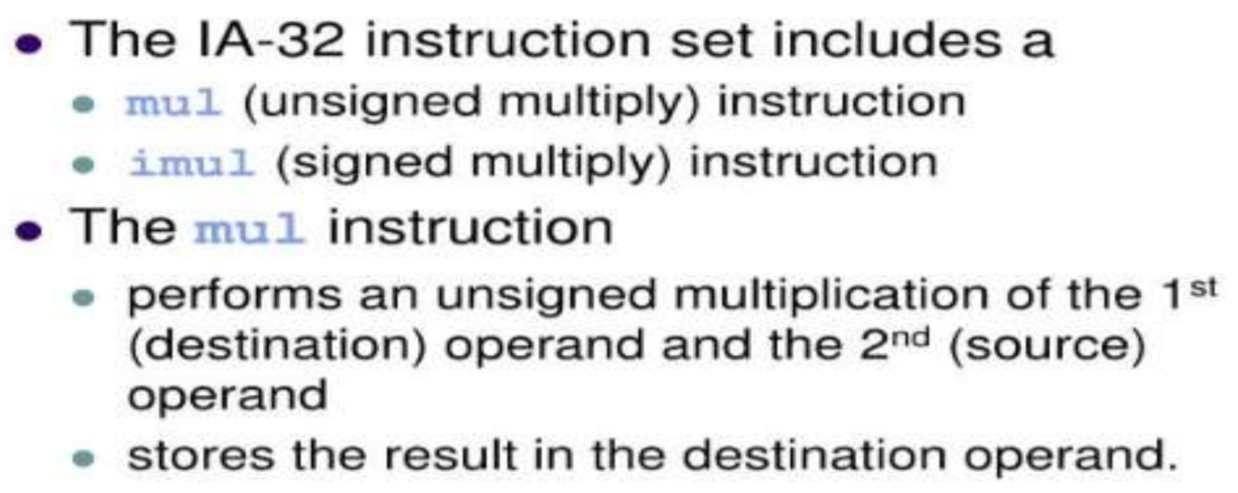
Signed/Unsigned Examples:-



- The max value for an unsigned integer is 2^n-1
  - $2^n-1 \times 2^n-1 = 2^{2n} - 2^{n+1} + 1 < 2^{2n}$
- The minimum value for a signed integer is -2^{n-1}
  - $-2^{n-1} \times -2^{n-1} = 2^{2n-2}\ 2 < 2^{2n}$

Integer Security

Multiplication Instructions:-



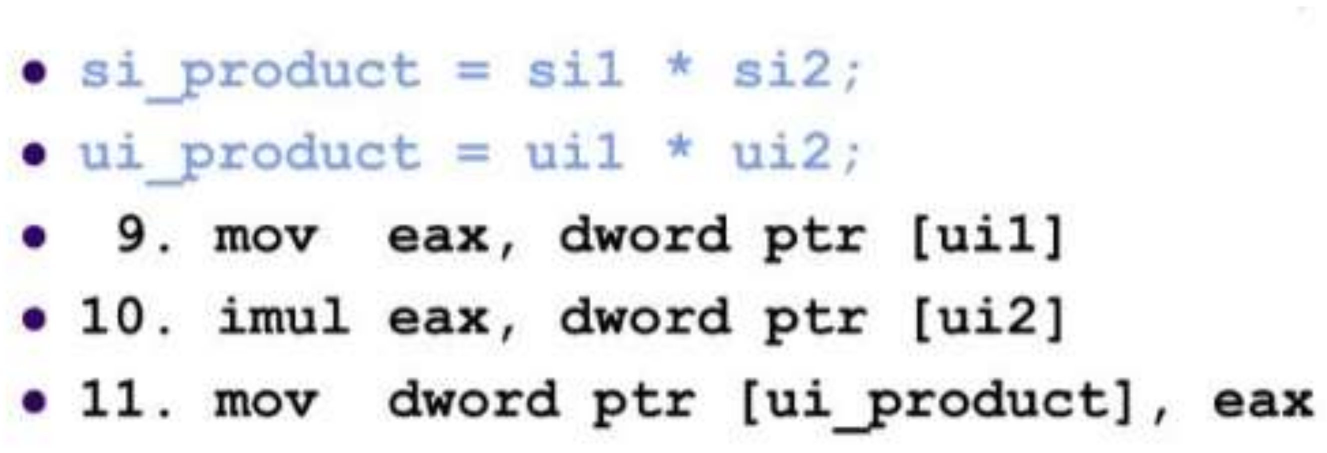
- The IA-32 instruction set includes a
  - `mul` (unsigned multiply) instruction
  - `imul` (signed multiply) instruction
- The `mul` instruction
  - performs an unsigned multiplication of the 1st (destination) operand and the 2nd (source) operand
  - stores the result in the destination operand.

Integer Security

Unsigned Multiplication :-

```
#include <stdio.h>

int main() {
    // Declare two unsigned integers
    unsigned int num1 = 123456789;
    unsigned int num2 = 987654321;

    // Perform multiplication
    unsigned int result = num1 * num2;

    // Print the result
    printf("Multiplication of %u and %u is: %u\n", num1, num2, result);

    return 0;
}
```

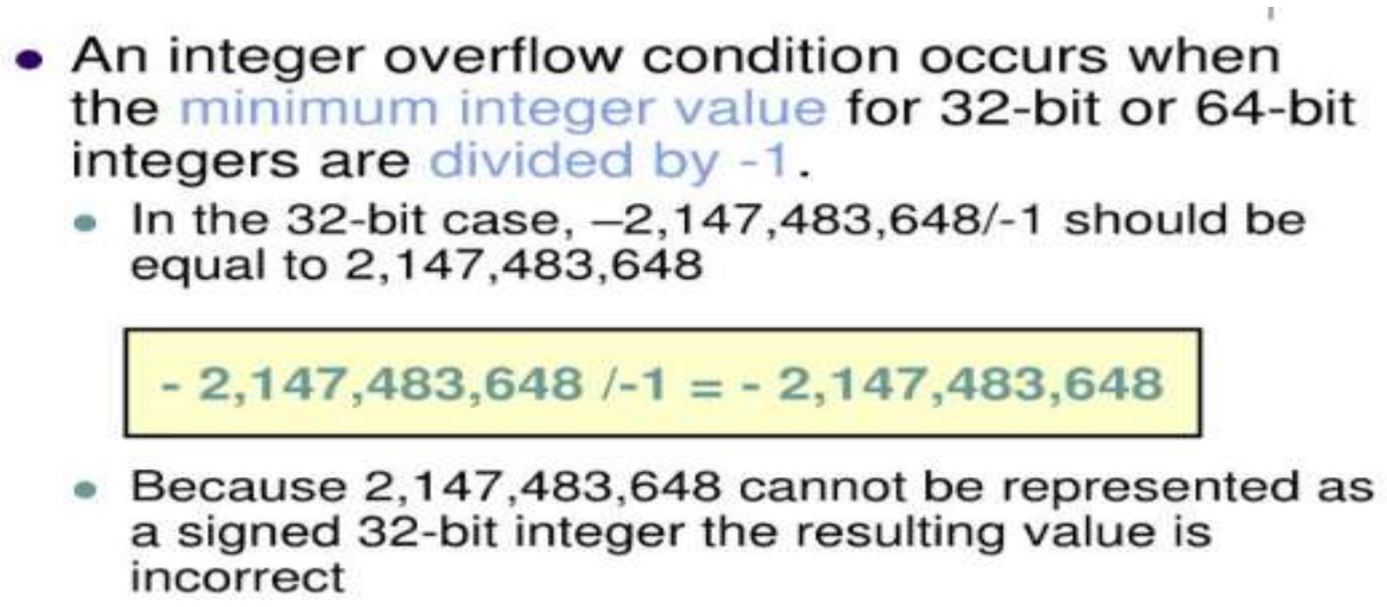
Signed/Unsigned int Multiplication :-



```
si_product = si1 * si2;
ui_product = ui1 * ui2;
mov   eax, dword ptr [ui1]
imul  eax, dword ptr [ui2]
mov   dword ptr [ui_product], eax
```

Integer Security

Integer Division :-



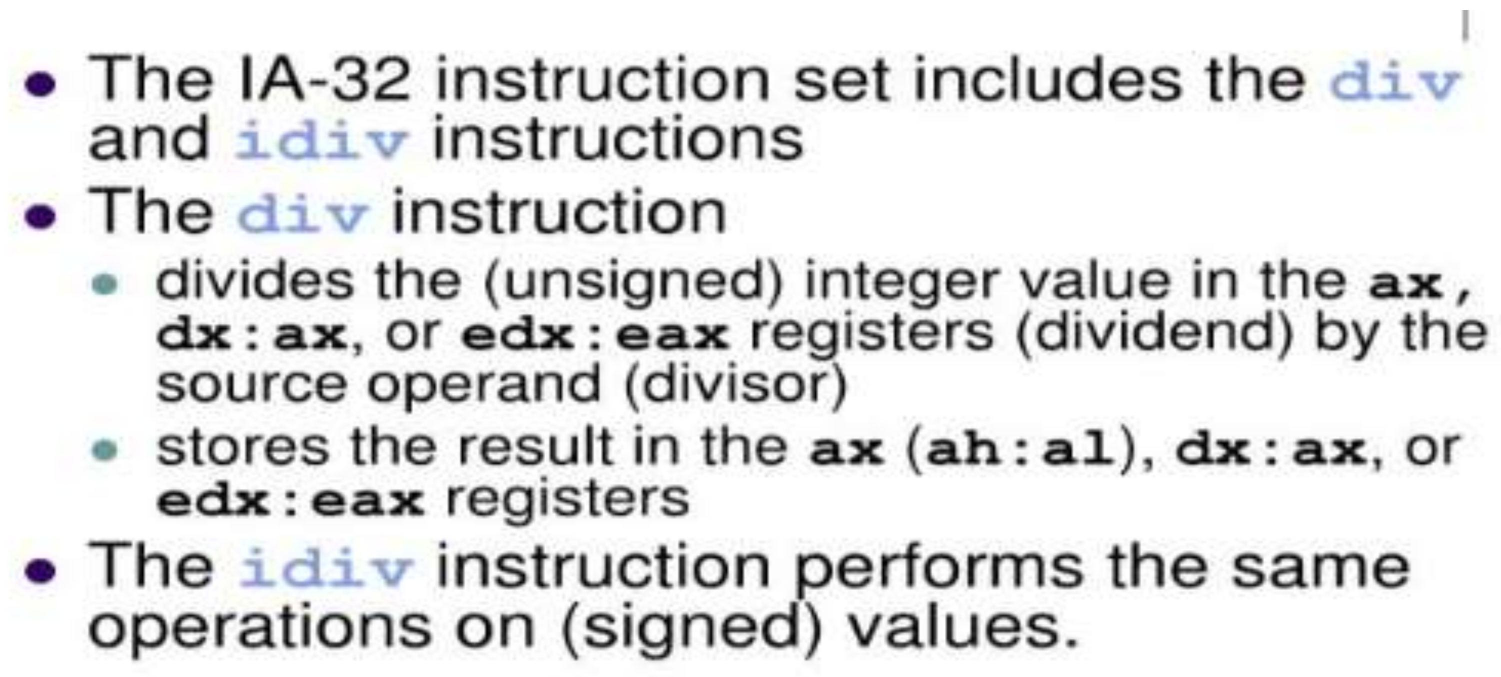
- An integer overflow condition occurs when the minimum integer value for 32-bit or 64-bit integers are divided by -1.
  - In the 32-bit case, –2,147,483,648/-1 should be equal to 2,147,483,648

    - 2,147,483,648 /-1 = - 2,147,483,648

  - Because 2,147,483,648 cannot be represented as a signed 32-bit integer the resulting value is incorrect

Integer Security

Error Detection :-



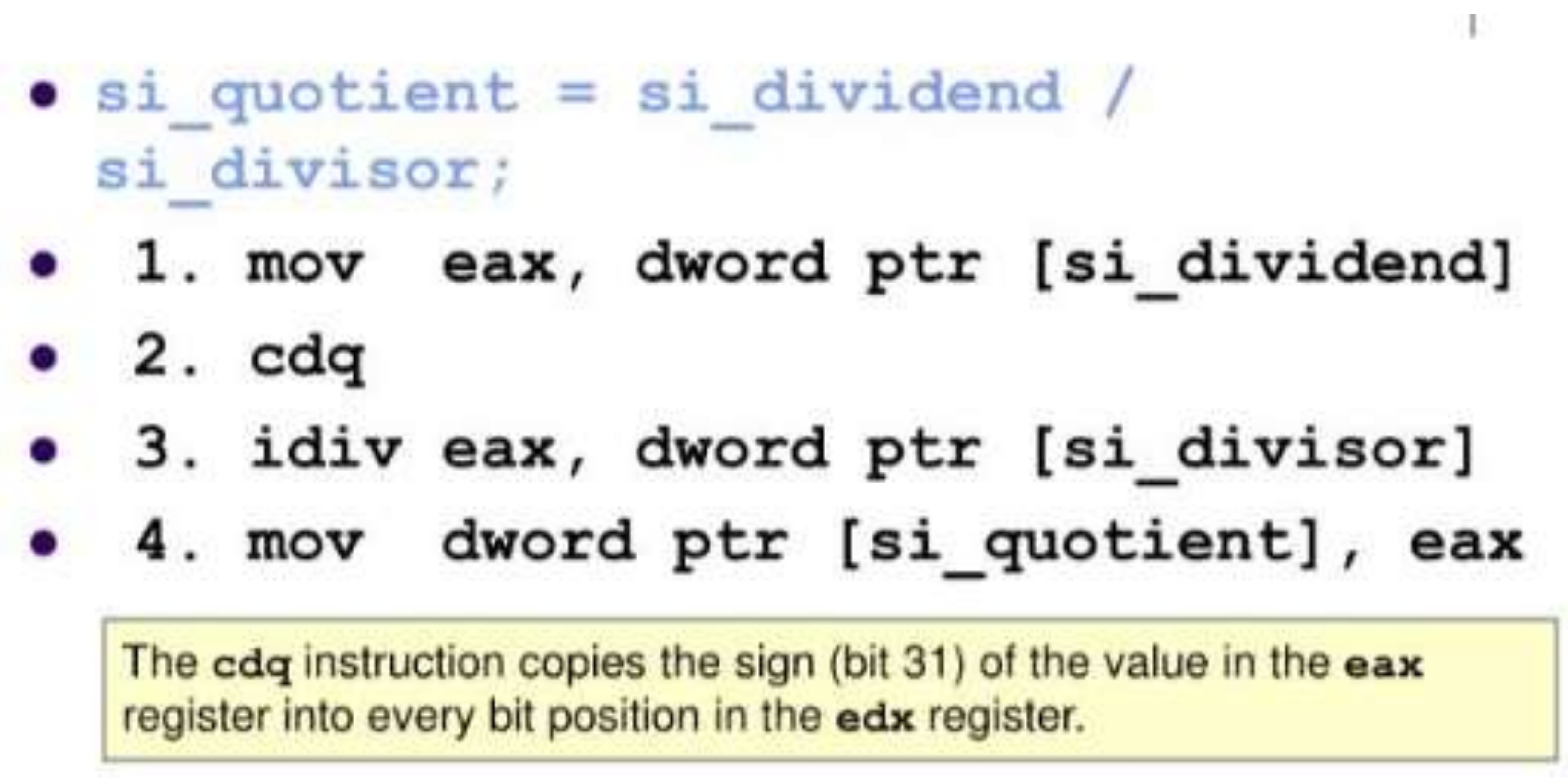
- The IA-32 instruction set includes the `div` and `idiv` instructions
- The `div` instruction
  - divides the (unsigned) integer value in the **`ax`**, **`dx:ax`**, or **`edx:eax`** registers (dividend) by the source operand (divisor)
  - stores the result in the **`ax`** (**`ah:al`**), **`dx:ax`**, or **`edx:eax`** registers
- The `idiv` instruction performs the same operations on (signed) values.

Integer Security-Signed Integer Division

```
#include <stdio.h>
int main() {
    int a = 10;
    int b = 3;
    int result;

    result = a / b;  // 10 / 3 = 3 (integer division truncates the decimal part)
    printf("10 / 3 = %d\n", result);

    a = -10;
    result = a / b;  // -10 / 3 = -3 (rounded towards zero)
    printf("-10 / 3 = %d\n", result);

    a = 10;
    b = -3;
    result = a / b;  // 10 / -3 = -3 (rounded towards zero)
    printf("10 / -3 = %d\n", result);

    a = -10;
    b = -3;
    result = a / b;  // -10 / -3 = 3 (both negative, result is positive)
    printf("-10 / -3 = %d\n", result);

    return 0;
}
```

Integer Security

Signed Integer Division

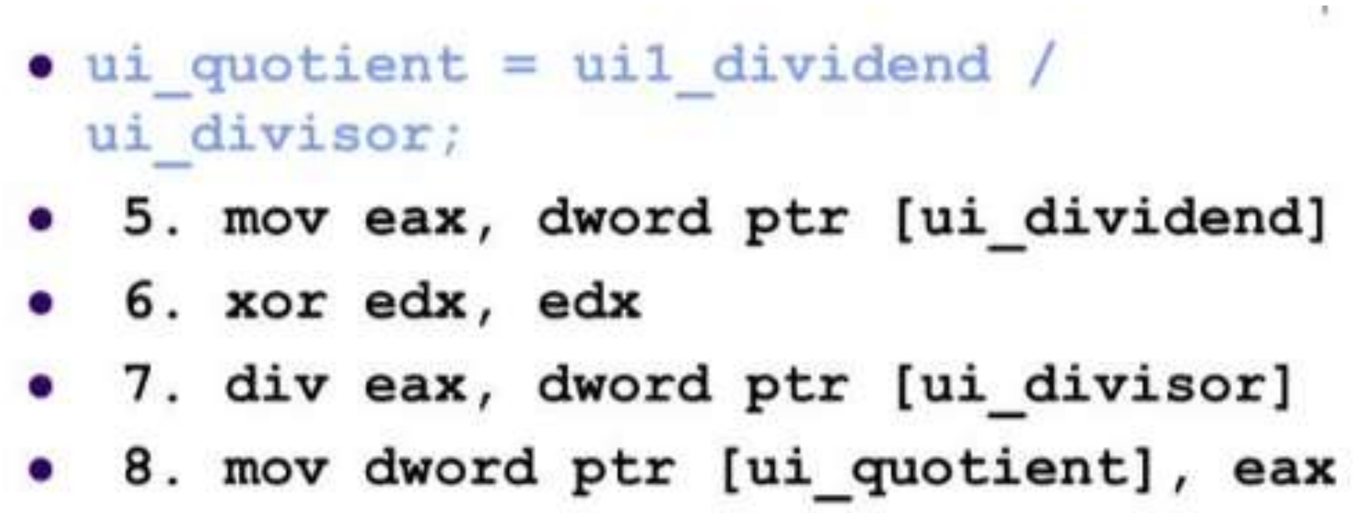


- `si_quotient = si_dividend / si_divisor;`
- `1. mov  eax, dword ptr [si_dividend]`
- `2. cdq`
- `3. idiv eax, dword ptr [si_divisor]`
- `4. mov  dword ptr [si_quotient], eax`

The **cdq** instruction copies the sign (bit 31) of the value in the **eax** register into every bit position in the **edx** register.

Integer Security

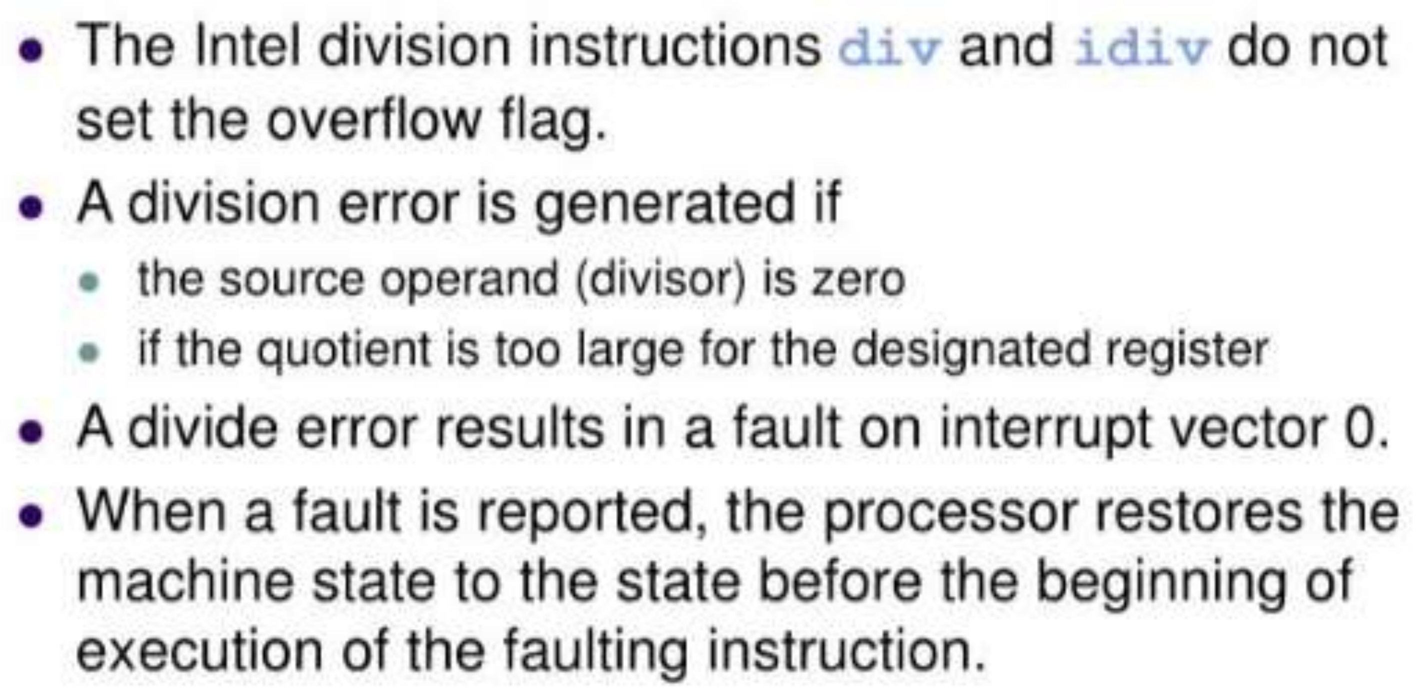
Unsigned Integer Division



- `ui_quotient = ui1_dividend / ui_divisor;`
- `5. mov eax, dword ptr [ui_dividend]`
- `6. xor edx, edx`
- `7. div eax, dword ptr [ui_divisor]`
- `8. mov dword ptr [ui_quotient], eax`

Integer Security

Error Detection



- The Intel division instructions `div` and `idiv` do not set the overflow flag.
- A division error is generated if
  - the source operand (divisor) is zero
  - if the quotient is too large for the designated register
- A divide error results in a fault on interrupt vector 0.
- When a fault is reported, the processor restores the machine state to the state before the beginning of execution of the faulting instruction.