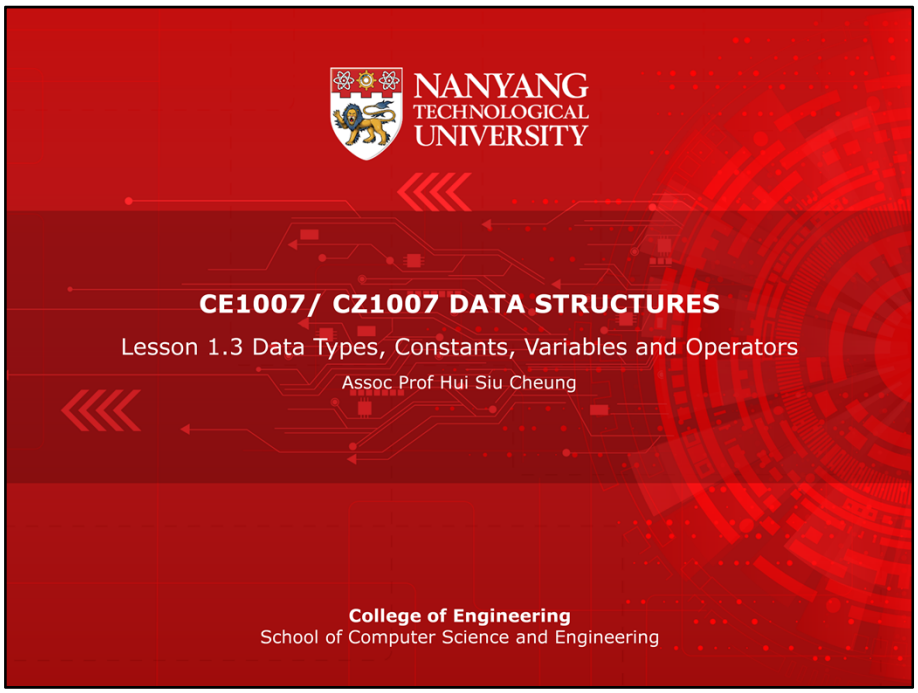
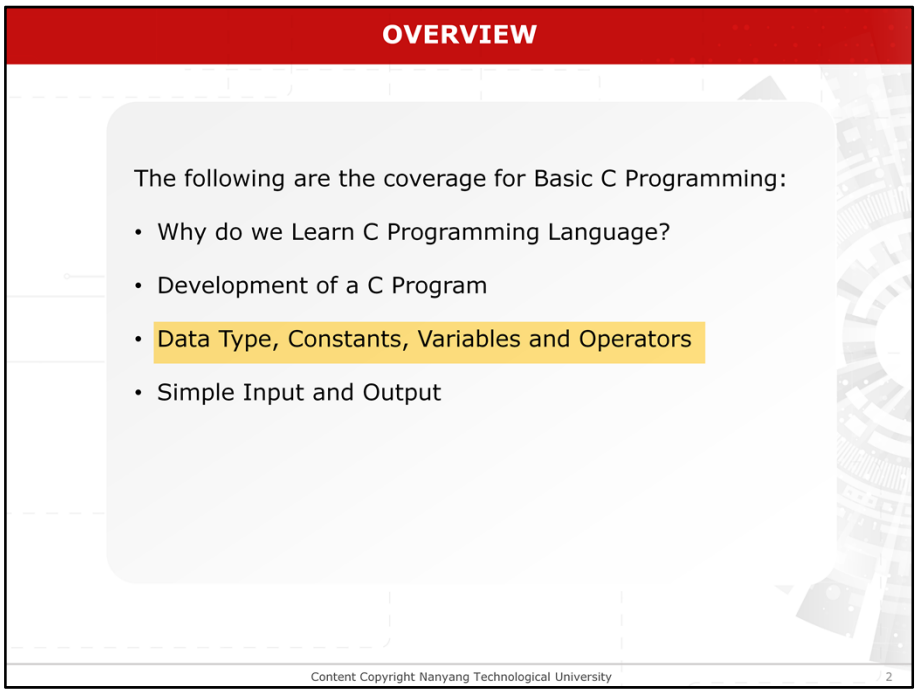


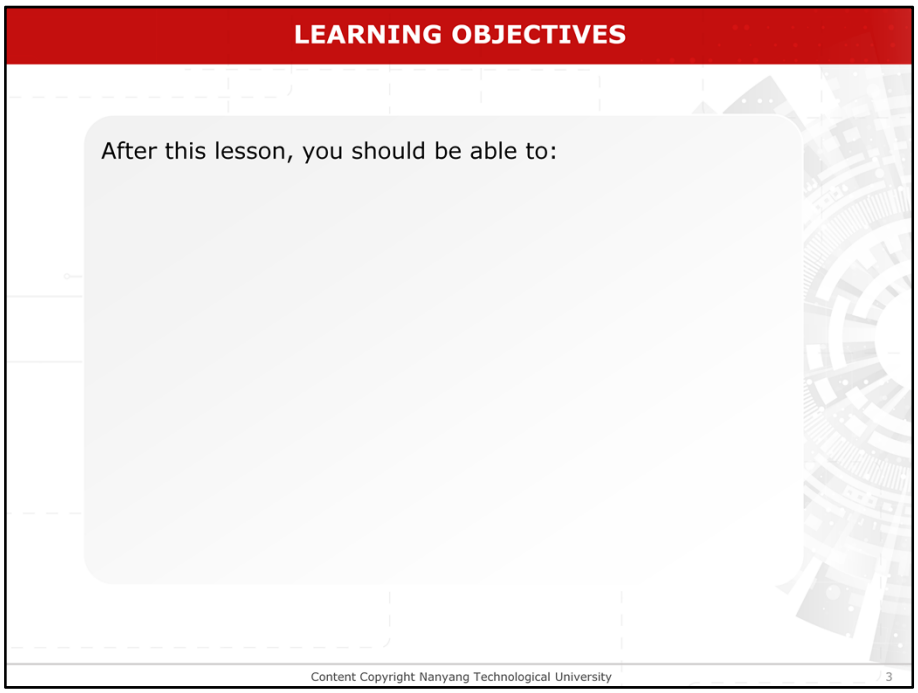
This lesson is on Basic C-Programming



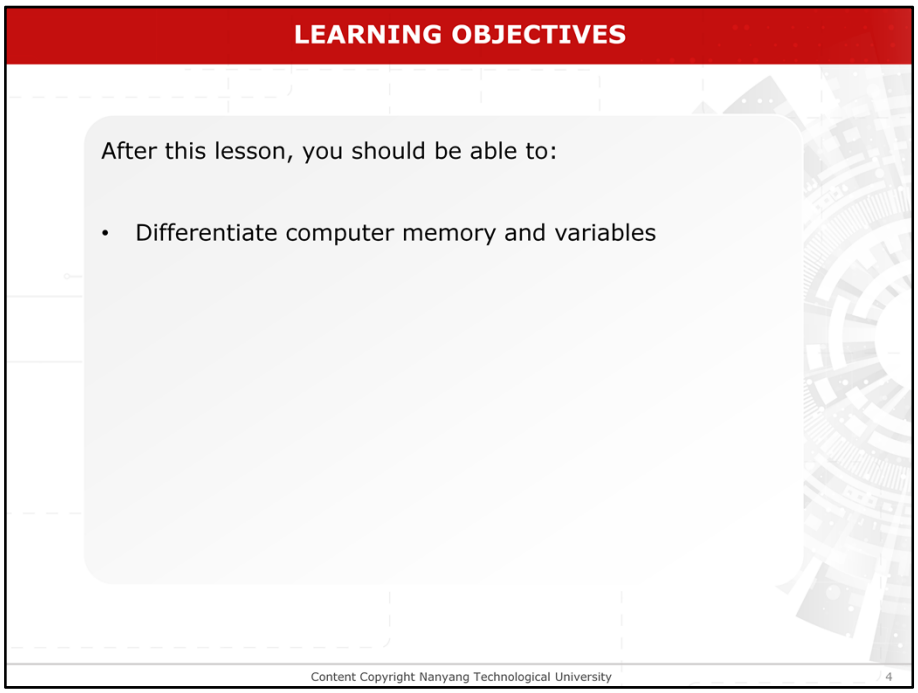
Basic C Programming

There are 4 main sections to cover for Basic C Programming as shown.

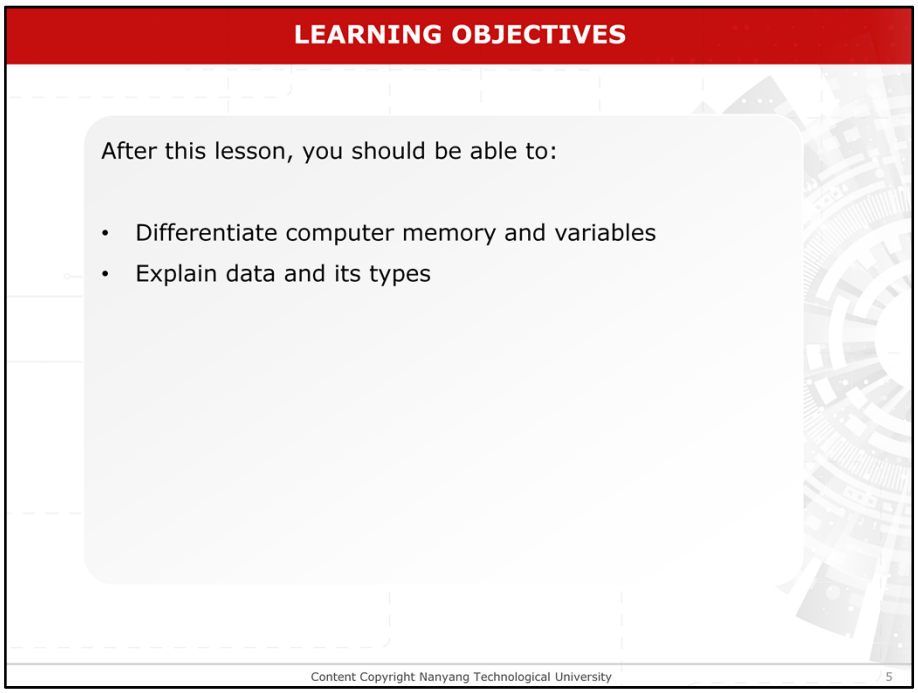
This video lesson focuses on the third part: Data types, constants, variables and operations.



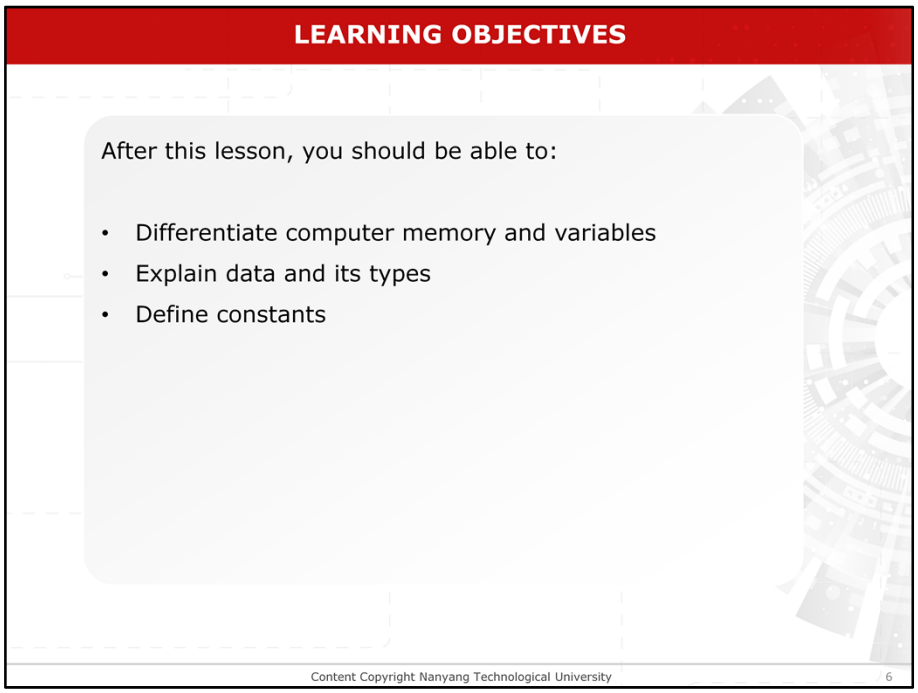
After this lesson you should be able to
[Pause 1 sec]



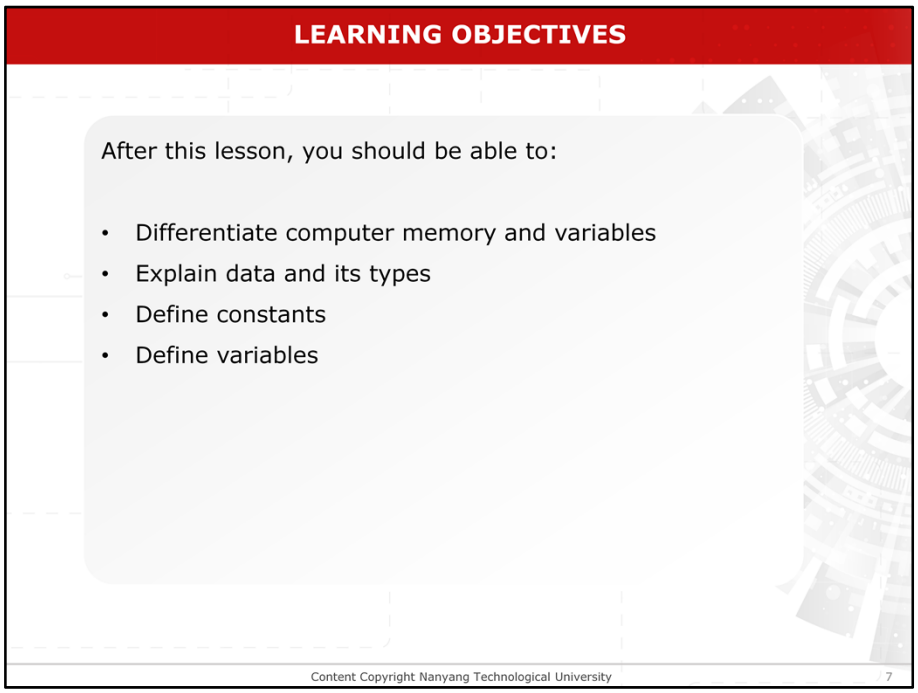
Differentiate computer memory and variables



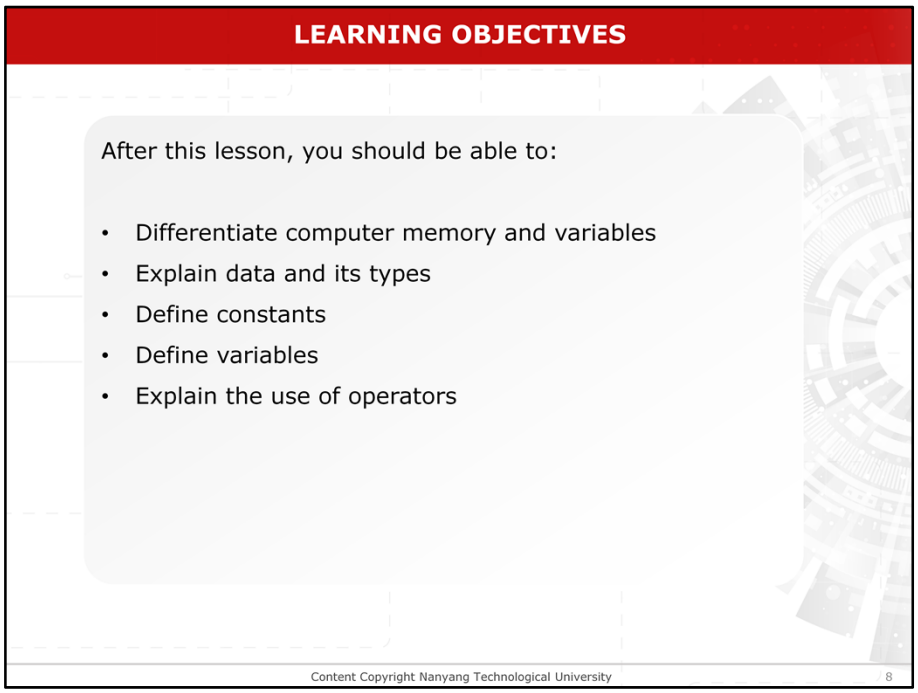
Explain Data and its types.
[Pause 1 sec]



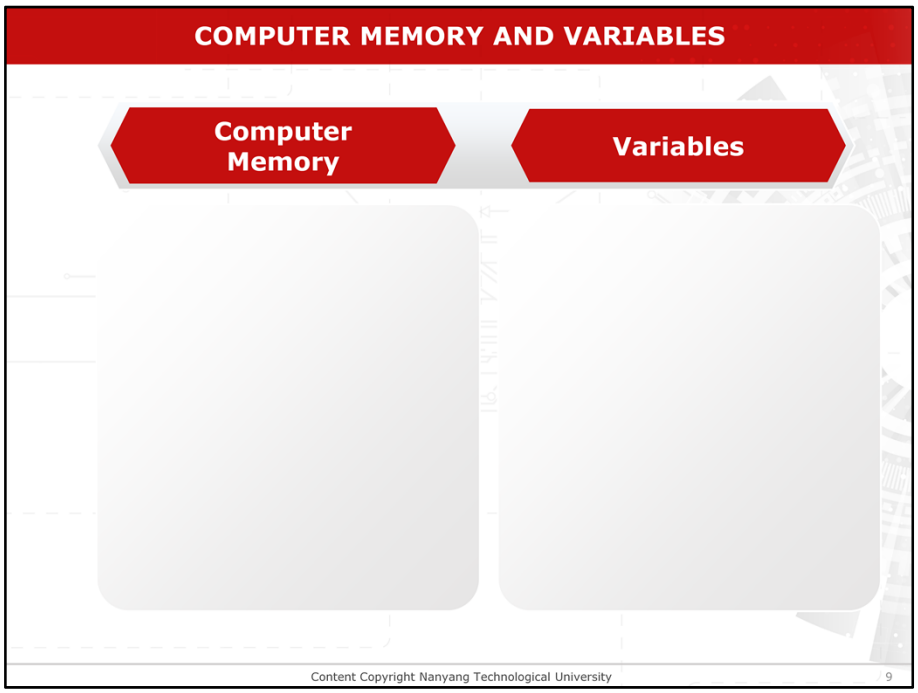
Define constants.
[Pause 1 sec]



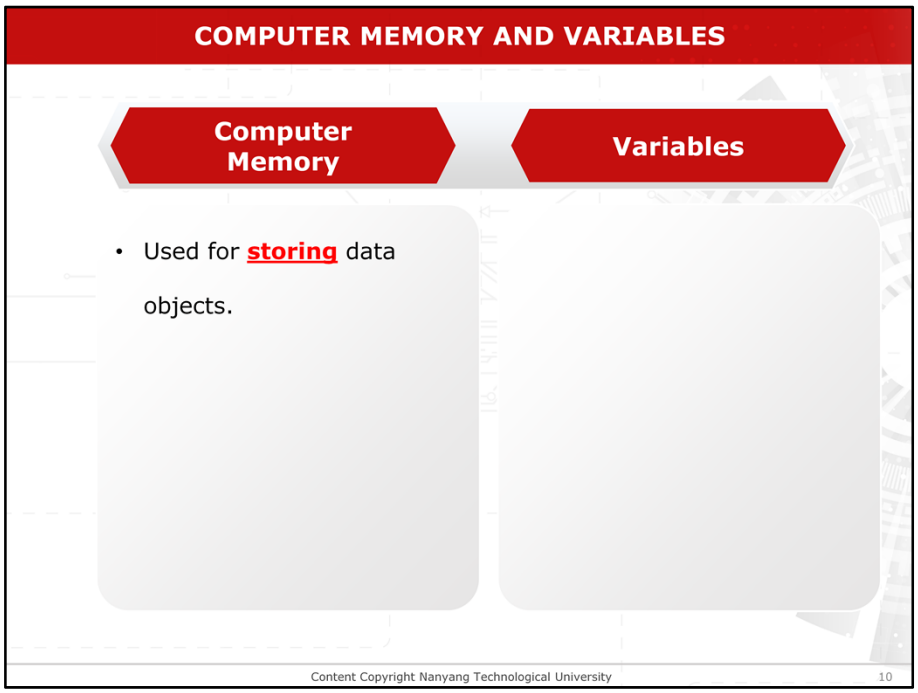
Define variables
[Pause 1 sec]



explain the use of operators.
[Pause 1 sec]

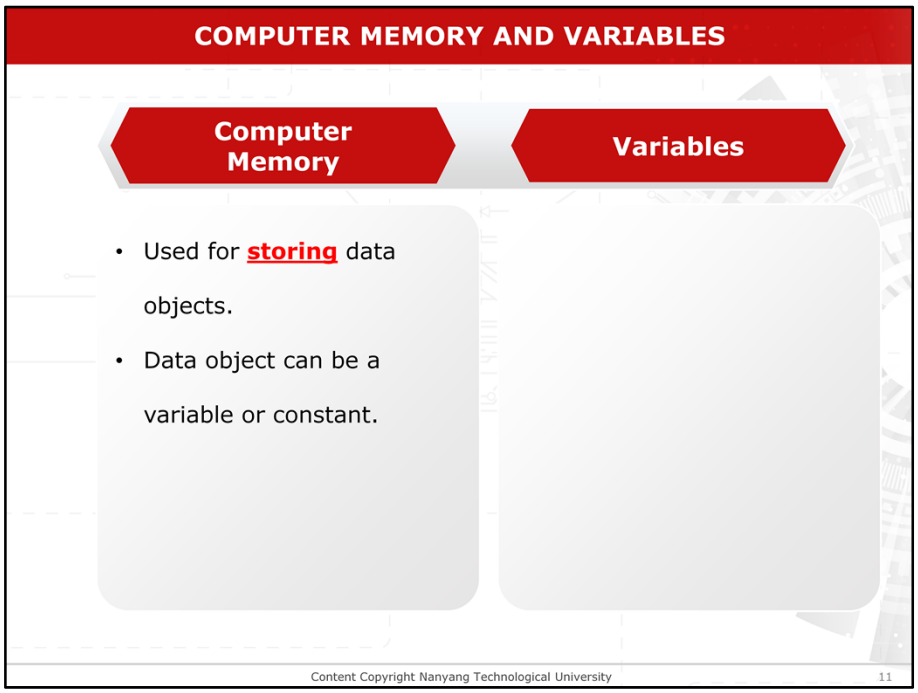


Computer Memory and Variables

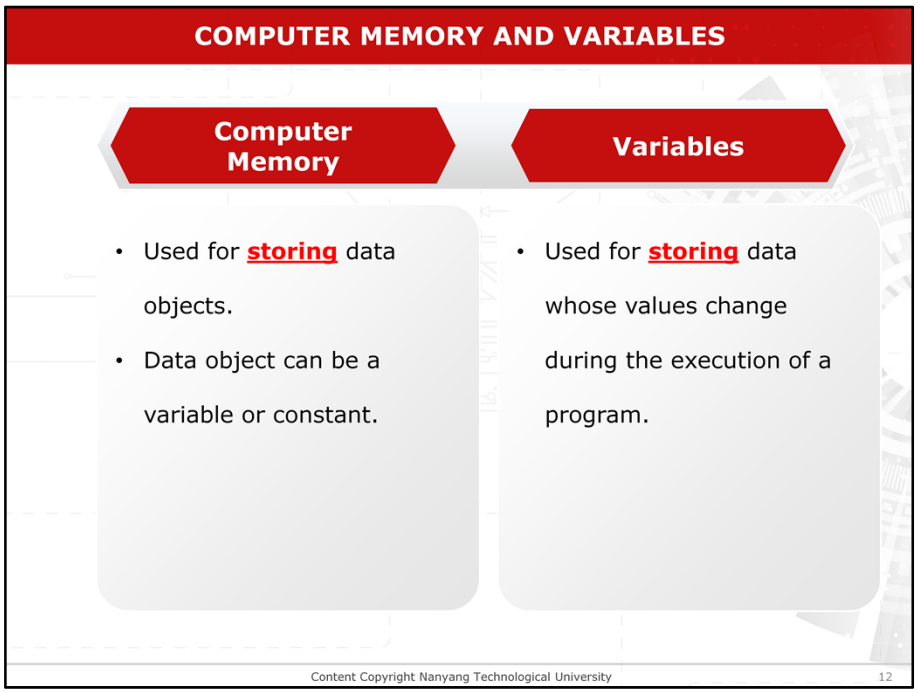


Computer Memory and Variables

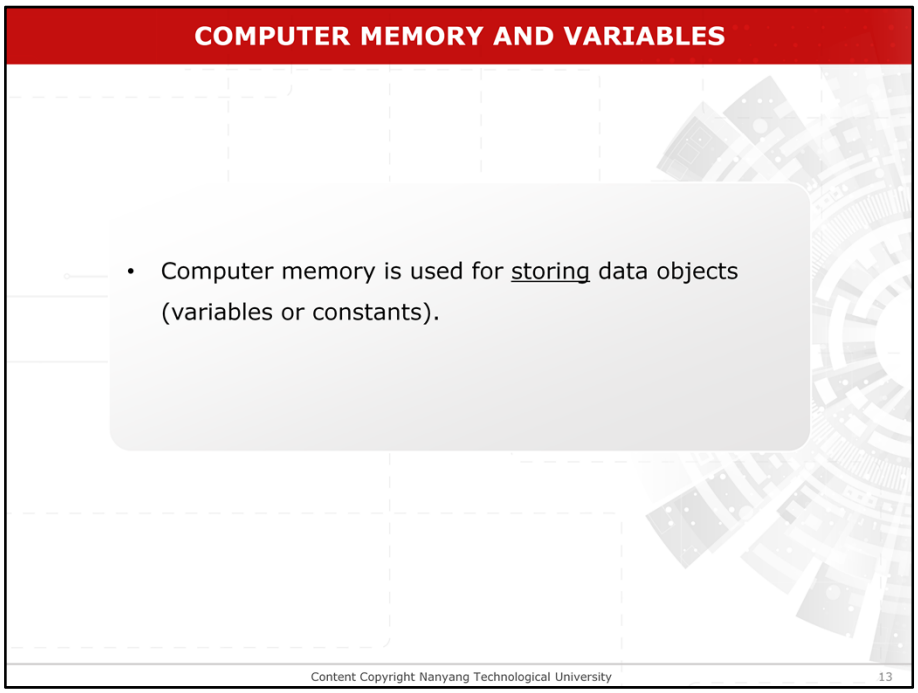
A program needs to work with *data* which are stored in the main memory.



A data object in a C program can be a variable or a constant.

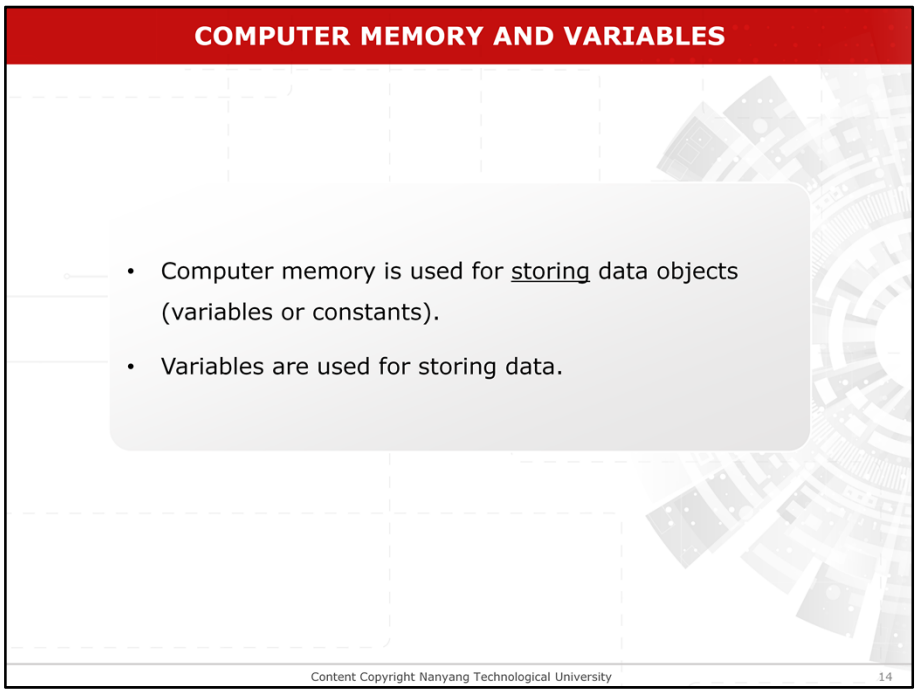


Variables are used to store data whose values change during the execution of a program, while constants can be used to store data whose values remain fixed during the execution of a program.



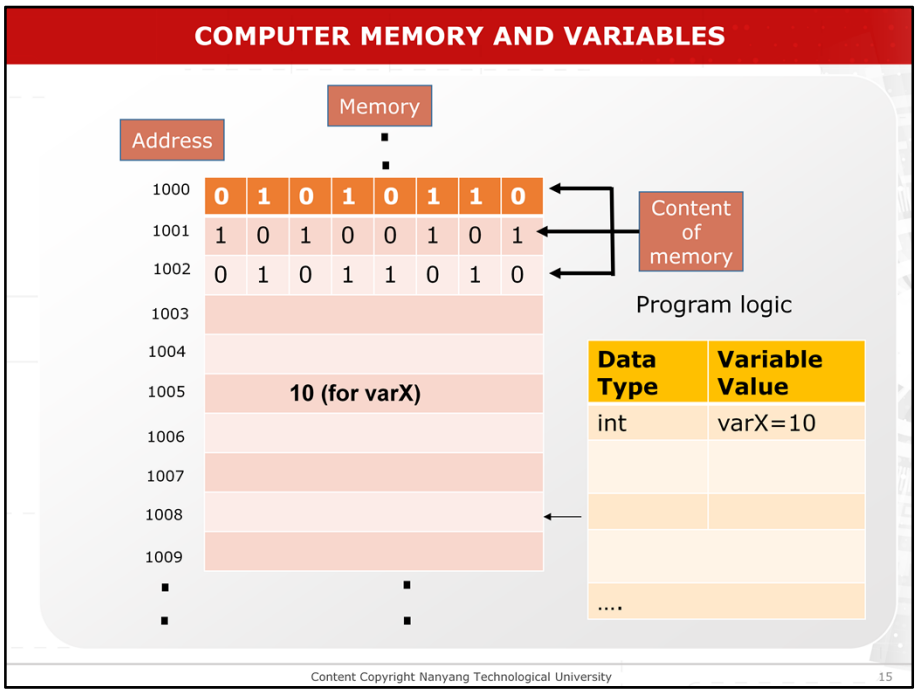
Computer Memory and Variables

Computer memory is used for storing data objects (variables or constants).



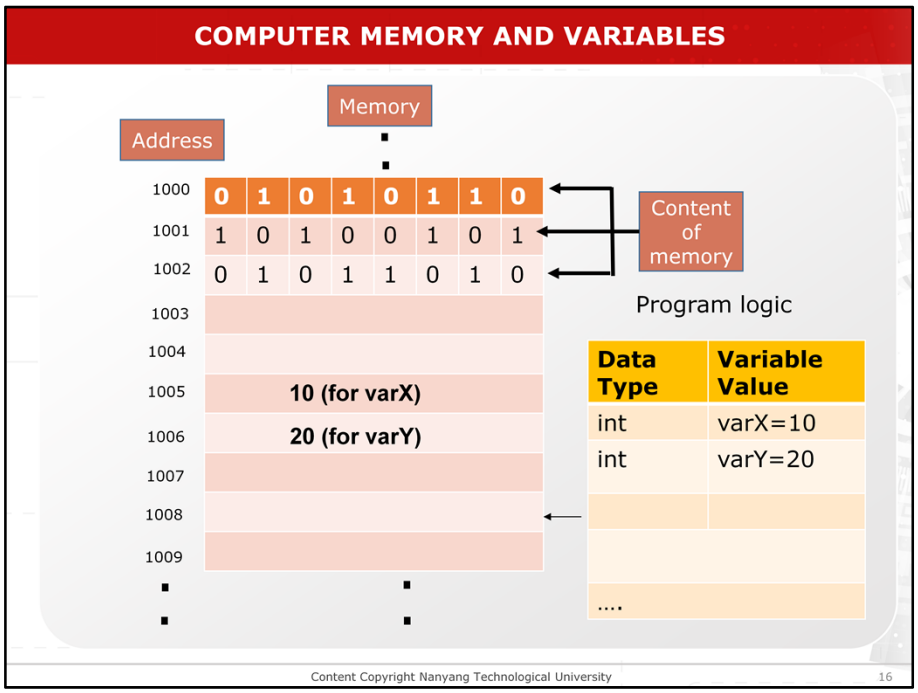
Computer Memory and Variables

Variables are used for storing data.

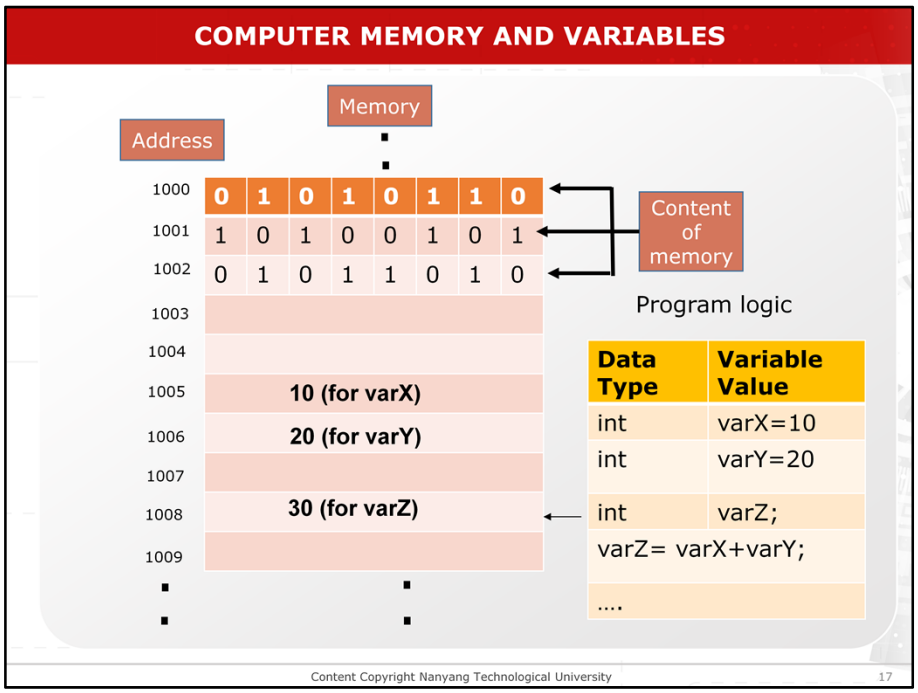


Computer Memory and Variables

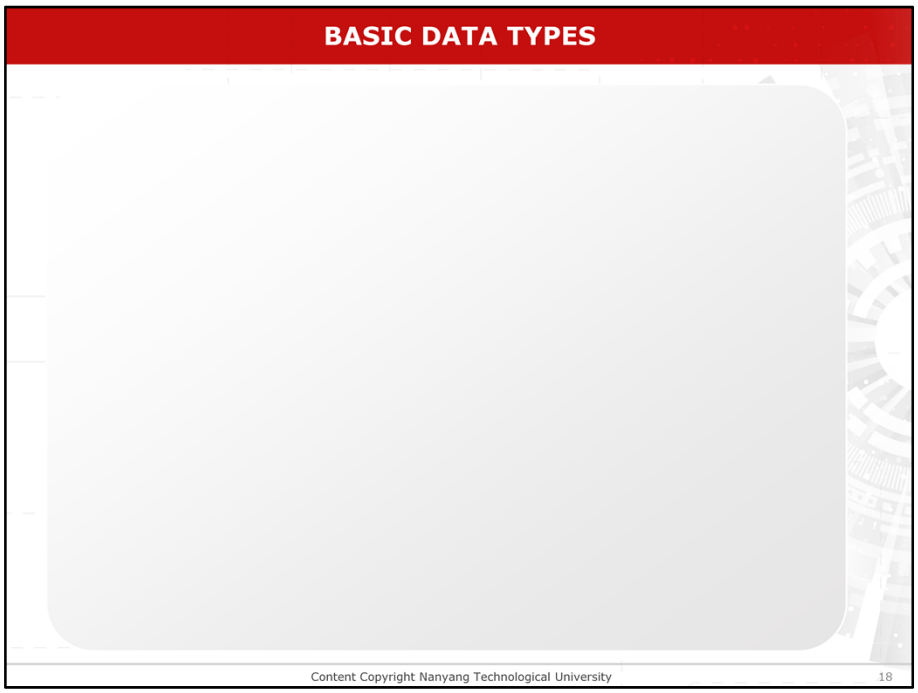
A program needs to work with *data* which are stored in the main memory



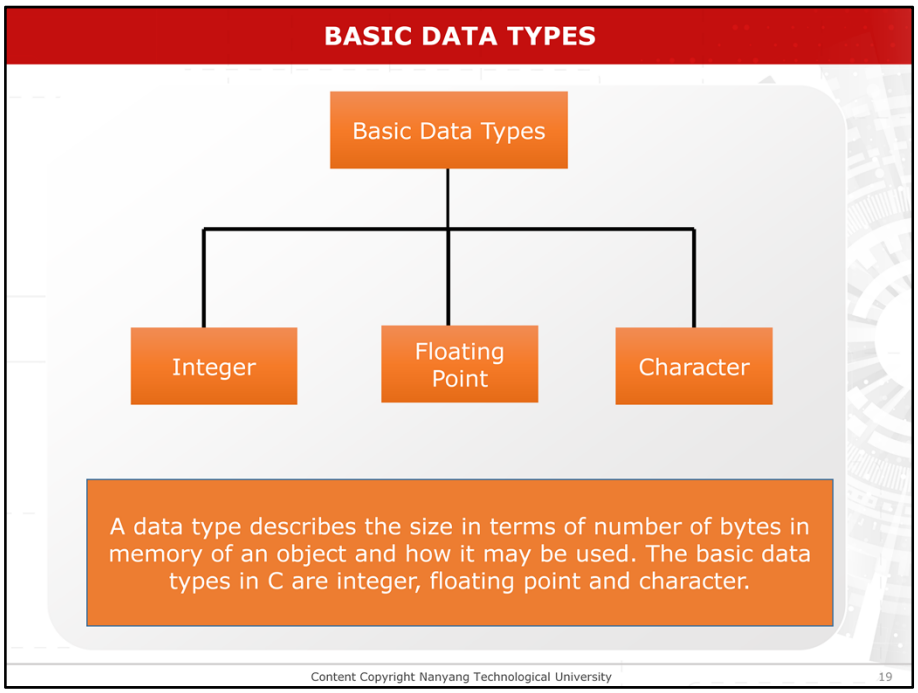
Program logic is shown here.



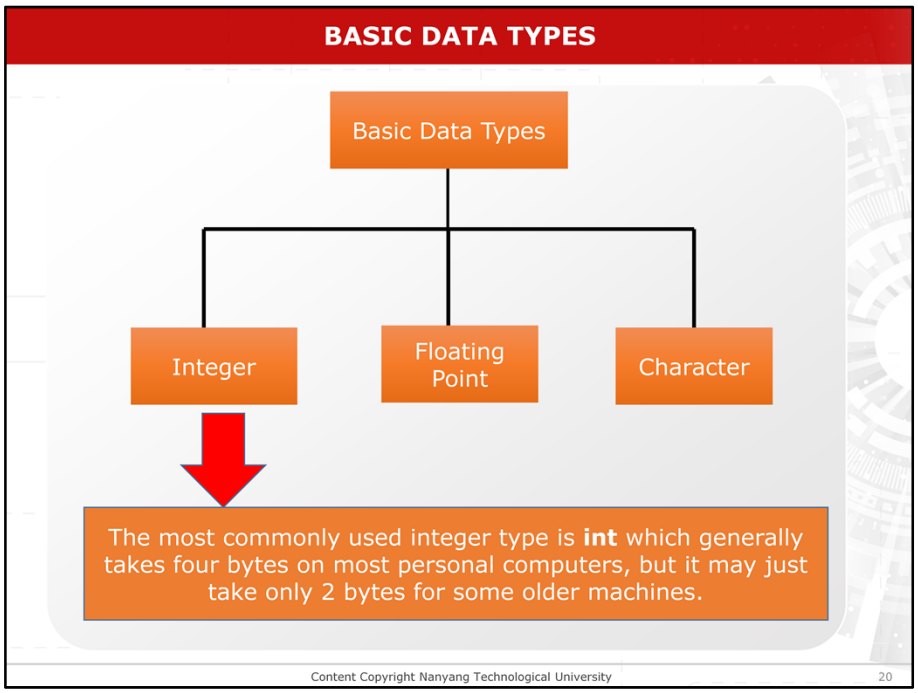
Both constants and variables must be of a certain type.



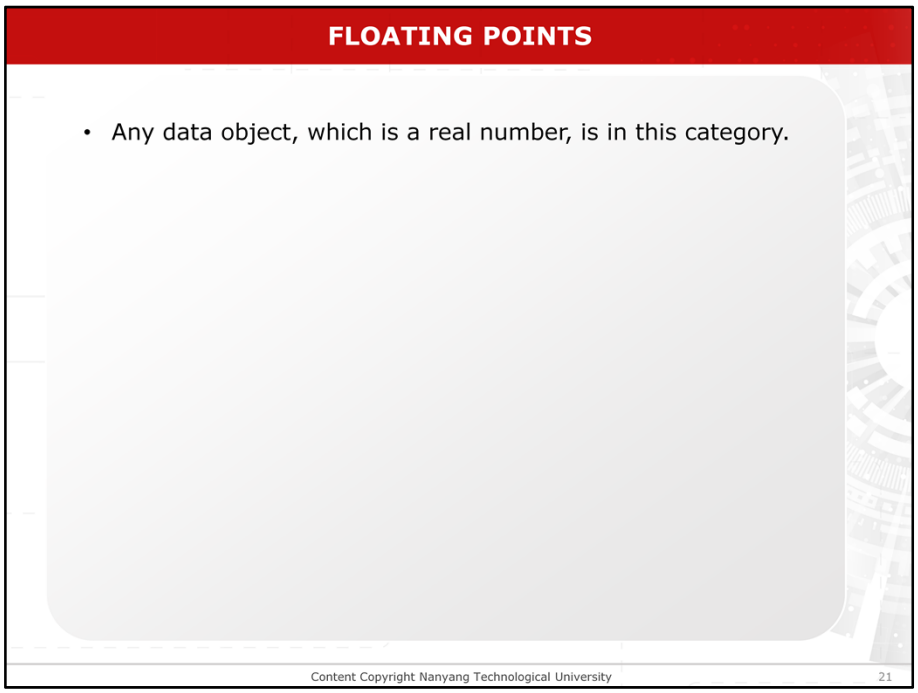
The C language provides a variety of data types for storing data for the needs of different programs.



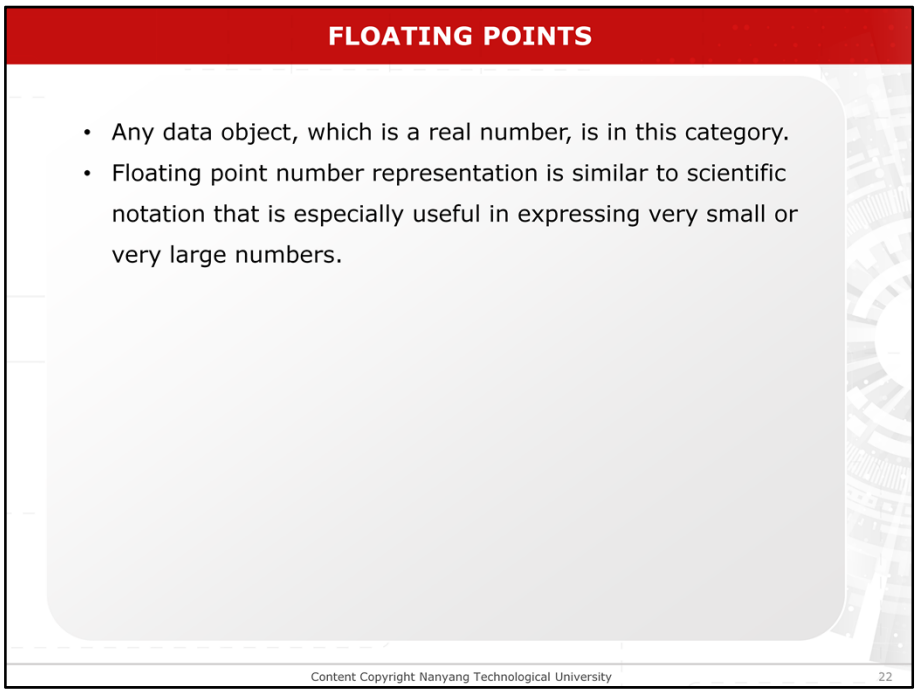
A data type describes the size in terms of number of bytes in memory of an object and how it may be used. The basic data types in C are integer, floating point and character. Each type has its own computational properties and memory requirements. Moreover, the same data type may take different sizes on different types of machines.



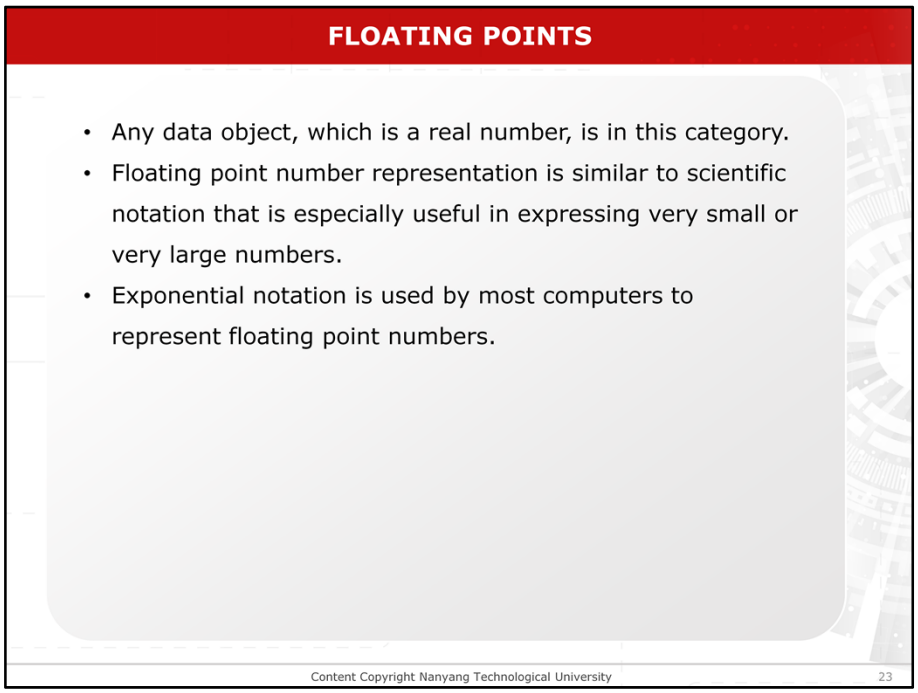
In C, there are six ways to store an integer object in a computer for C: short, int, long, unsigned, unsigned short, unsigned long. The most commonly used integer data type is **int** which generally takes four bytes on most personal computers, but it may just take only 2 bytes for some older machines.



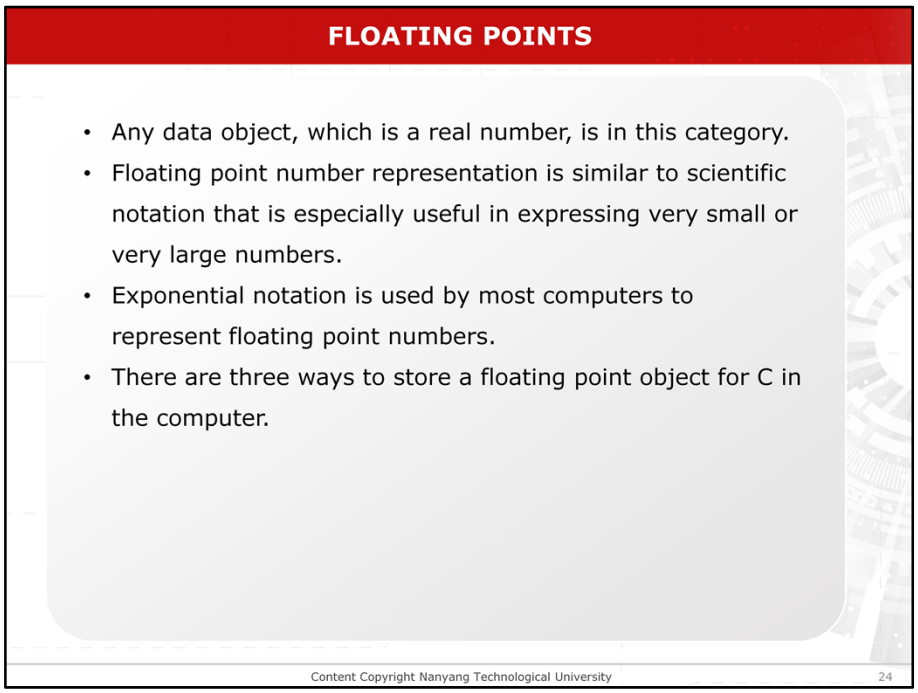
Any data object, which is a real number, is in this category.



Floating point number representation is similar to scientific notation that is especially useful in expressing very small or very large numbers.

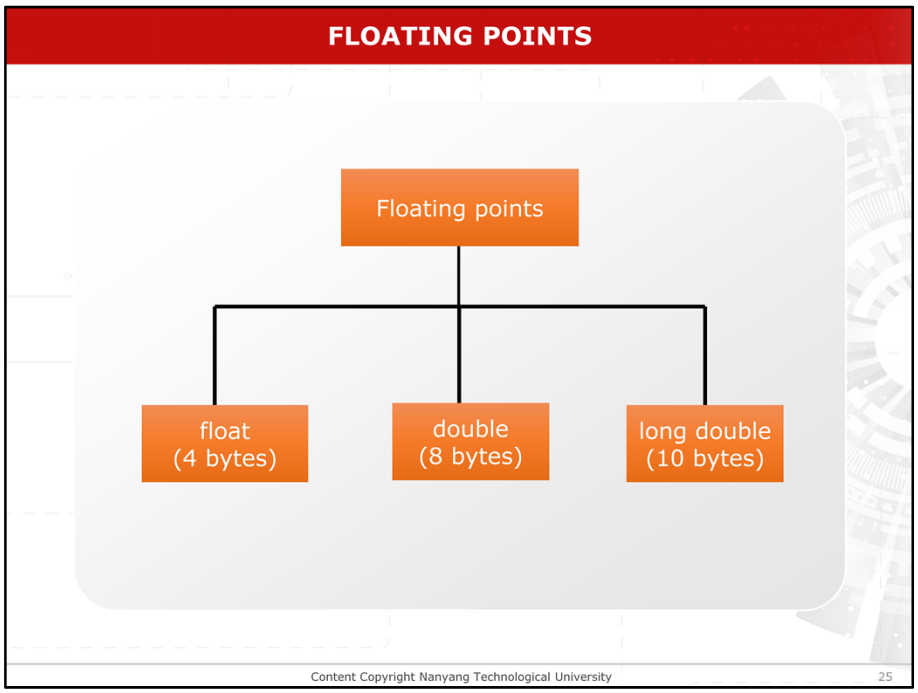


Exponential notation is used by most computers to represent floating point numbers.

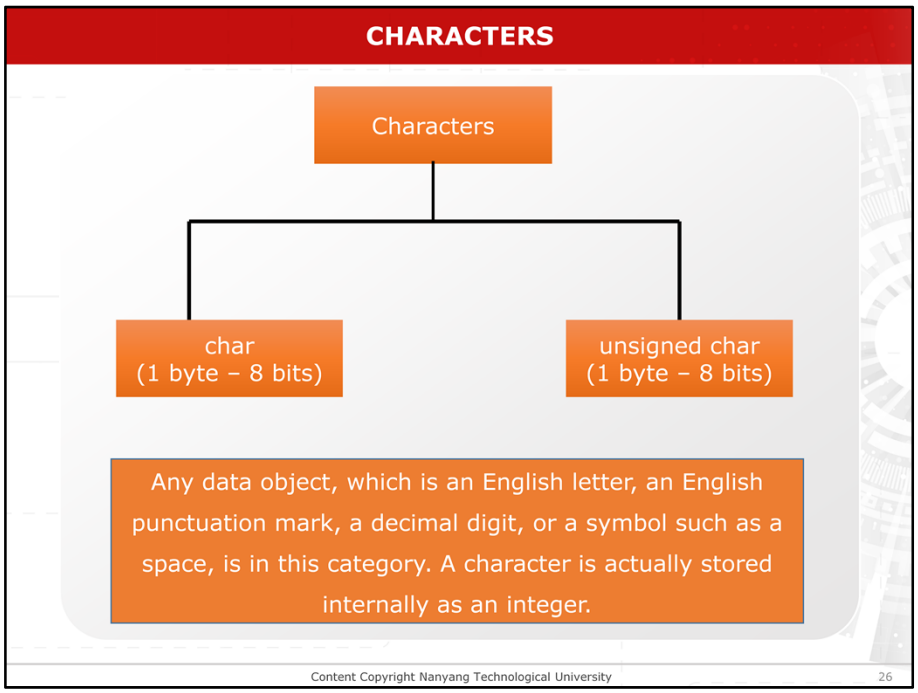


Floating Point Type

There are three ways to store a floating point object for C in the computer.

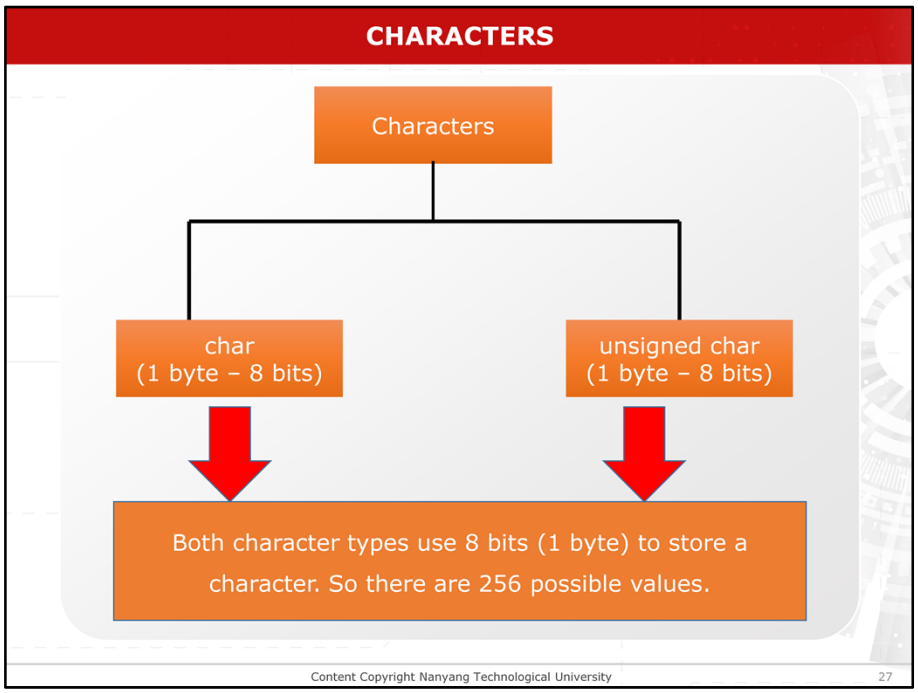


Therefore, three C floating point types are available: float, double and long double.



Character Type

Any data object, which is an English letter, an English punctuation mark, a decimal digit, or a symbol such as a space, is in this category. A character is actually stored internally as an integer.



Character Type

There are two C character types: **car** and **unsigned car**. Both character types use 8 bits (1 byte) to store a character. So there are 256 possible values.

CHARACTER - ASCII SET (1 BYTE)

	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

A character is actually stored internally as an integer. The conversion from characters to integer numbers is done commonly using the ASCII code. The ASCII (American Standard Code Information Interchange) code is the most commonly used character set. There are 128 distinct characters in the ASCII character set.

CHARACTER - ASCII SET (1 BYTE)

	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Characters

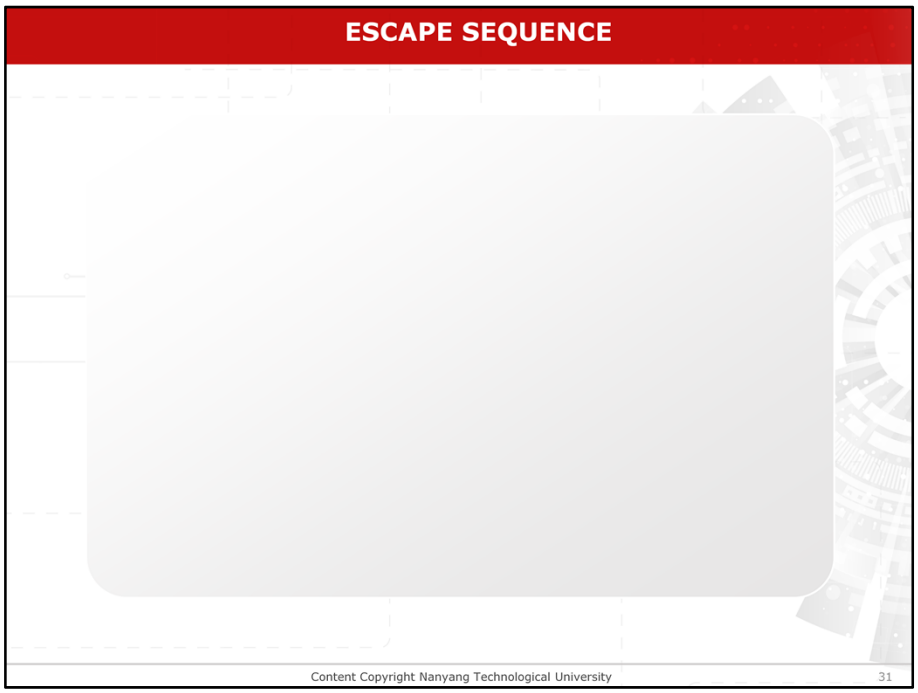
Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **A** in the ASCII character set, or by enclosing it with quotes, e.g. '**A**'.

CHARACTER - ASCII SET (1 BYTE)

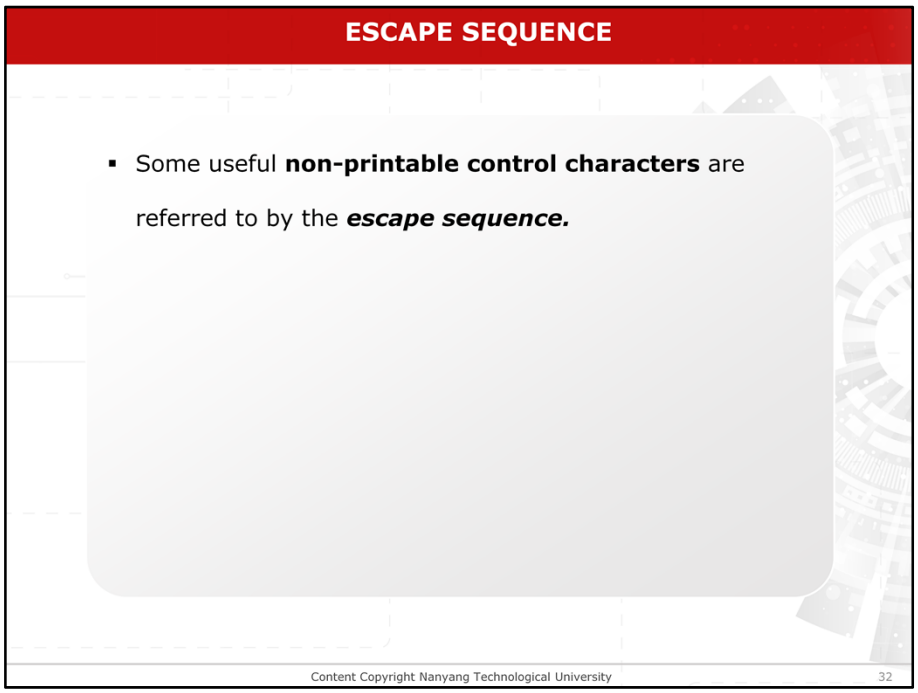
	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Characters

A *string* constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant.

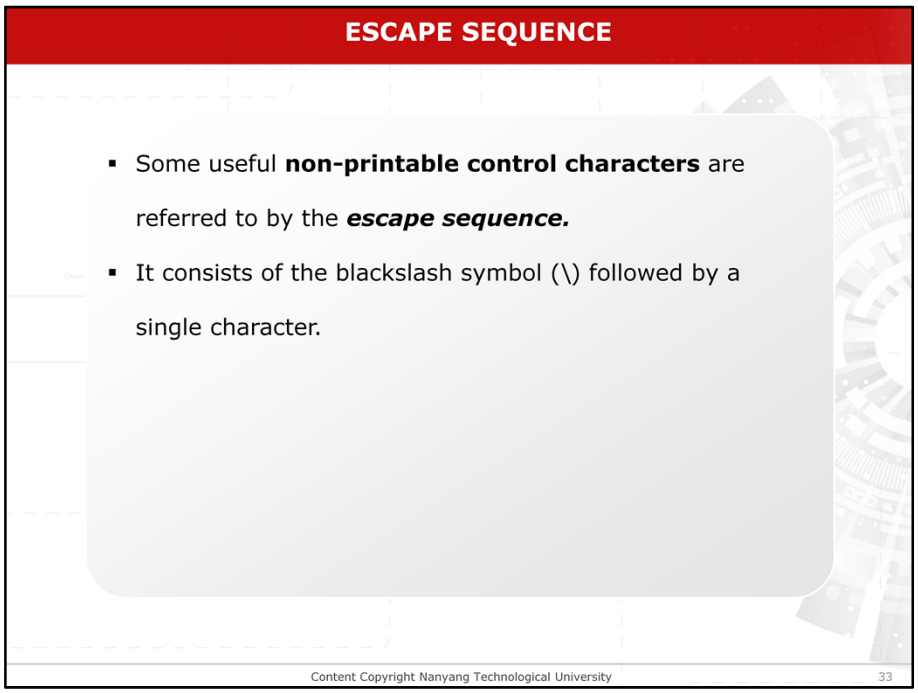


Escape Sequence

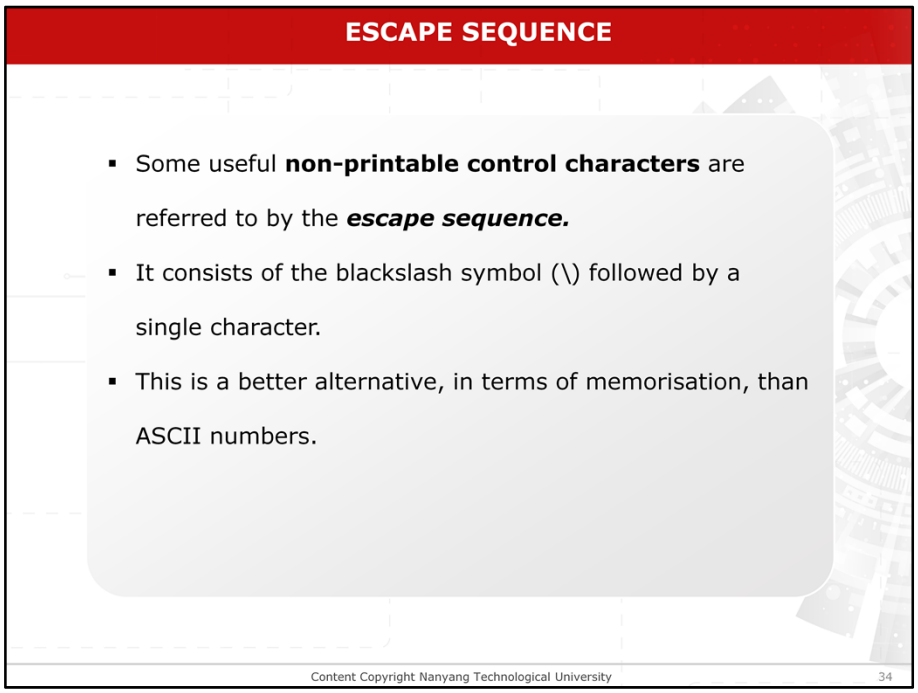


Characters – Escape Sequence

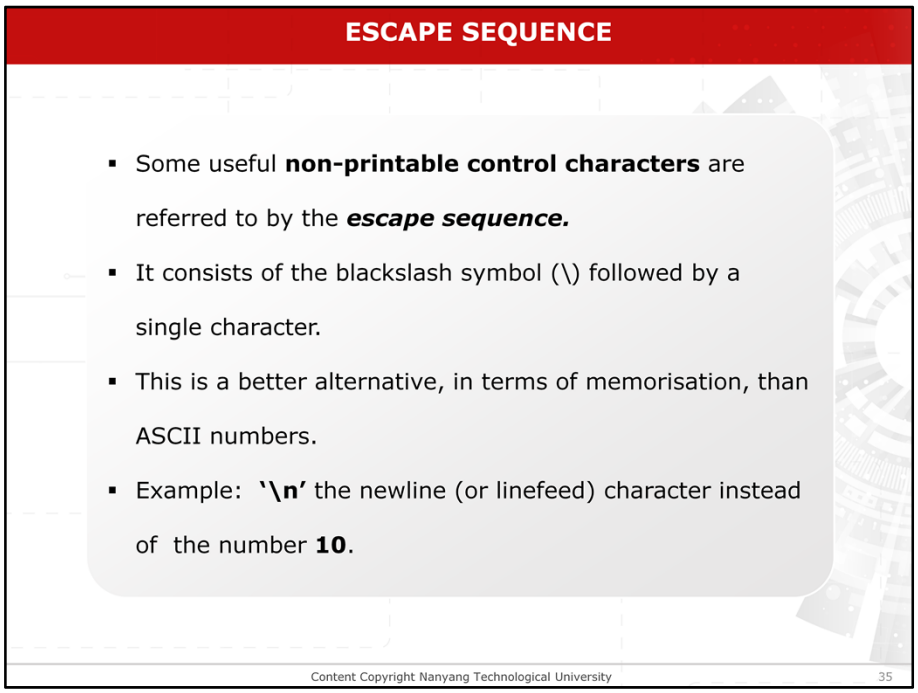
Some useful non-printable control characters are referred to by the *escape sequence.*



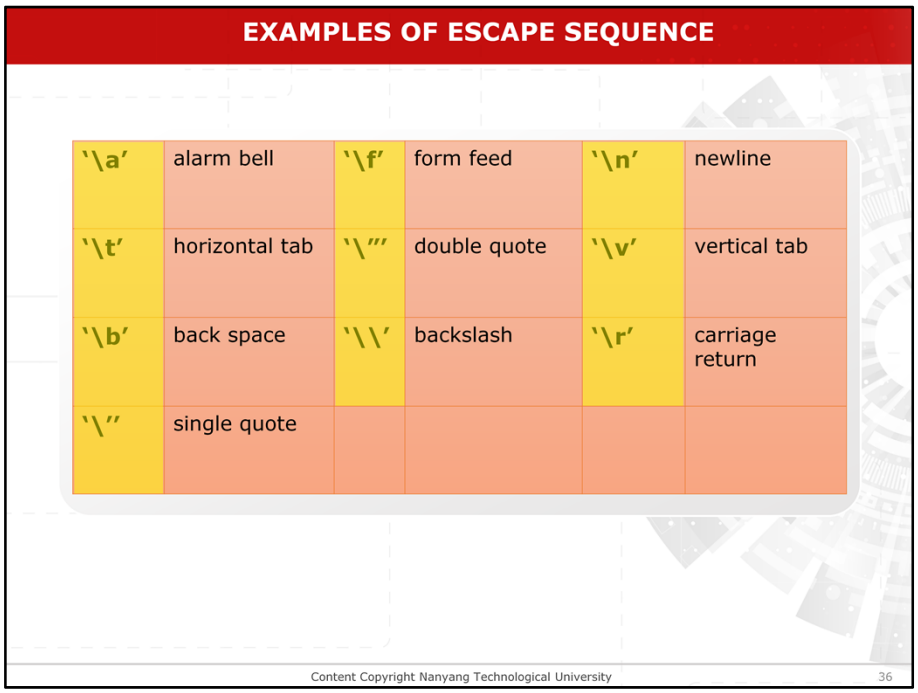
It consists of the backslash symbol followed by a single character. The two symbols together represent the single required character.



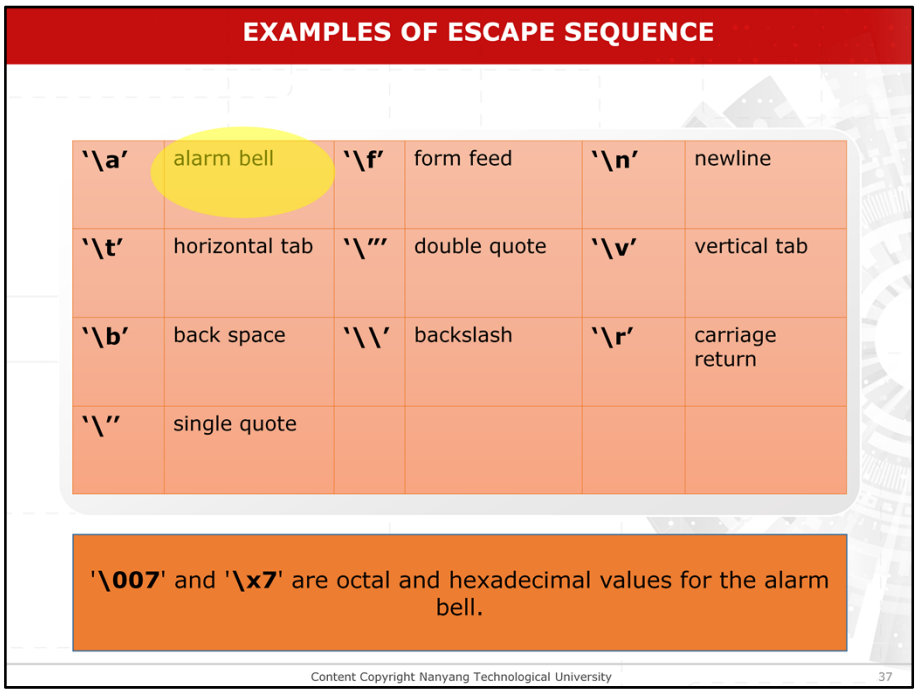
This is a better alternative, in terms of memorization, than ASCII numbers.



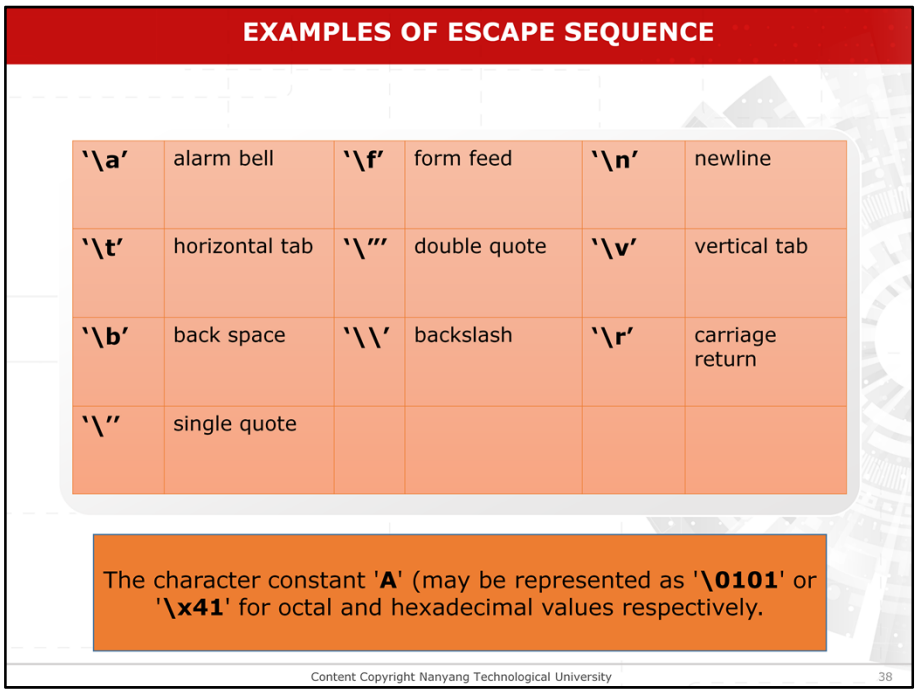
For example, '**\n**' represents the newline character instead of using the ASCII number 10.



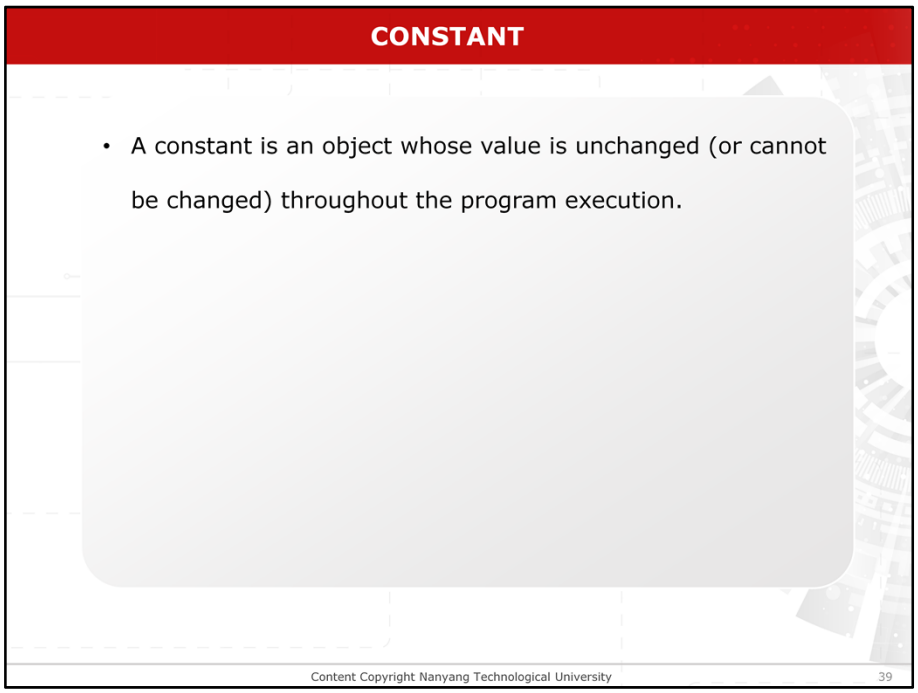
This table gives some examples of escape sequence. Notice that escape sequences must be enclosed in single quotes.



The formats for octal and hexadecimal values are '**\0oo**' and '**\xhh**'. For example, '**\007**' and '**\x7**' are octal and hexadecimal values for the alarm bell.

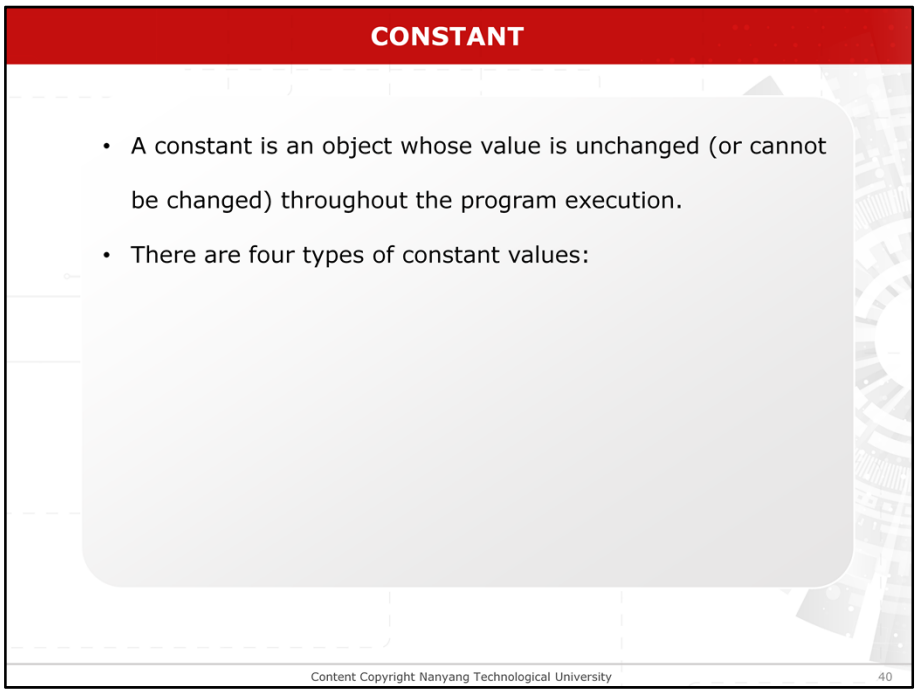


The character constant '**A**' (with ASCII encoding of decimal 65, octal 101 or hexadecimal 41) may be represented as '**\0101**' or '**\x41**' for octal and hexadecimal values respectively.

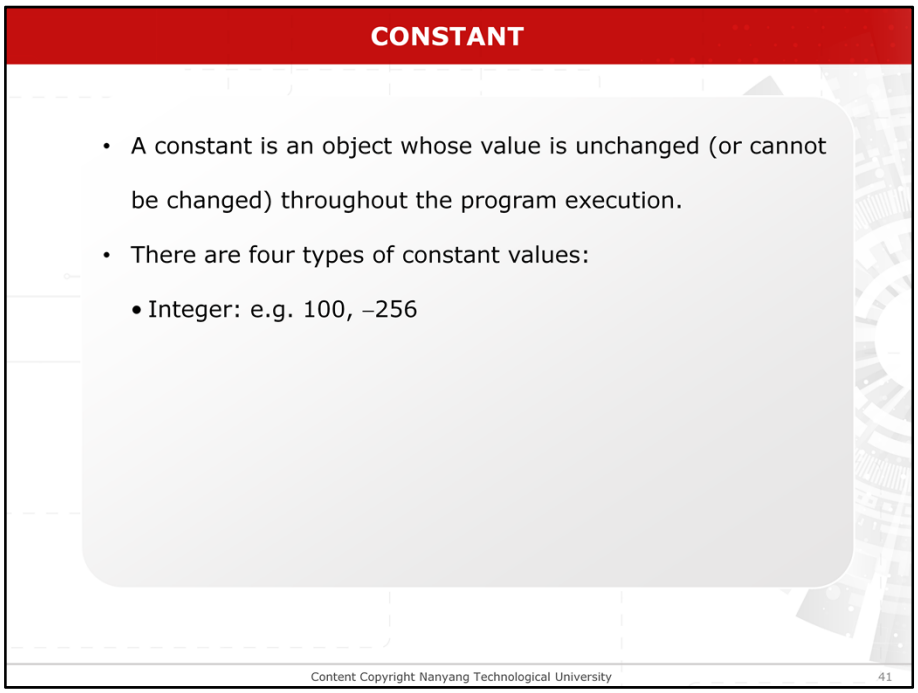


Constants

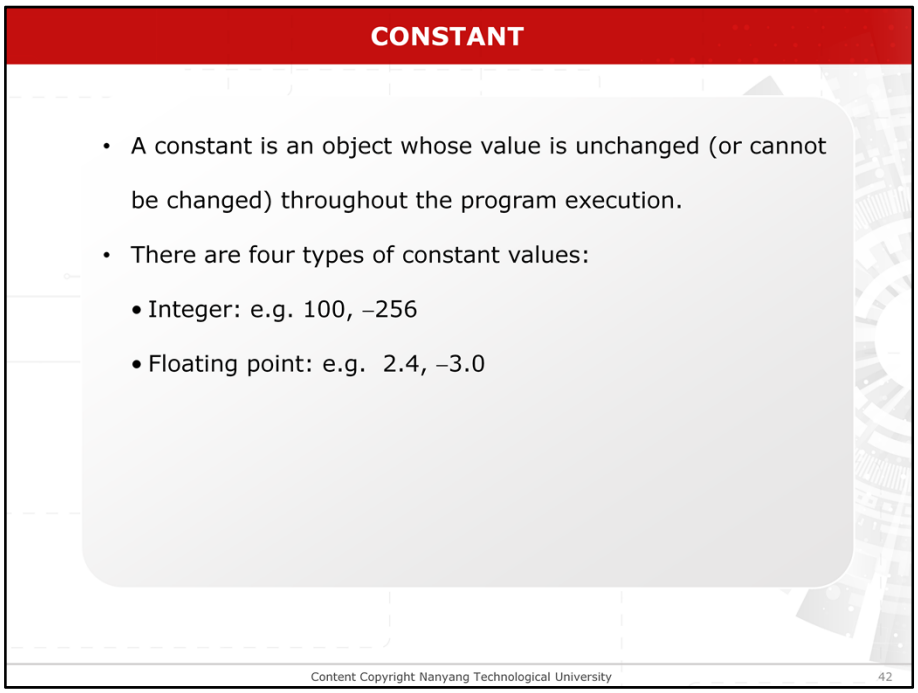
A constant is an object whose value is unchanged (or cannot be changed) throughout the program execution.



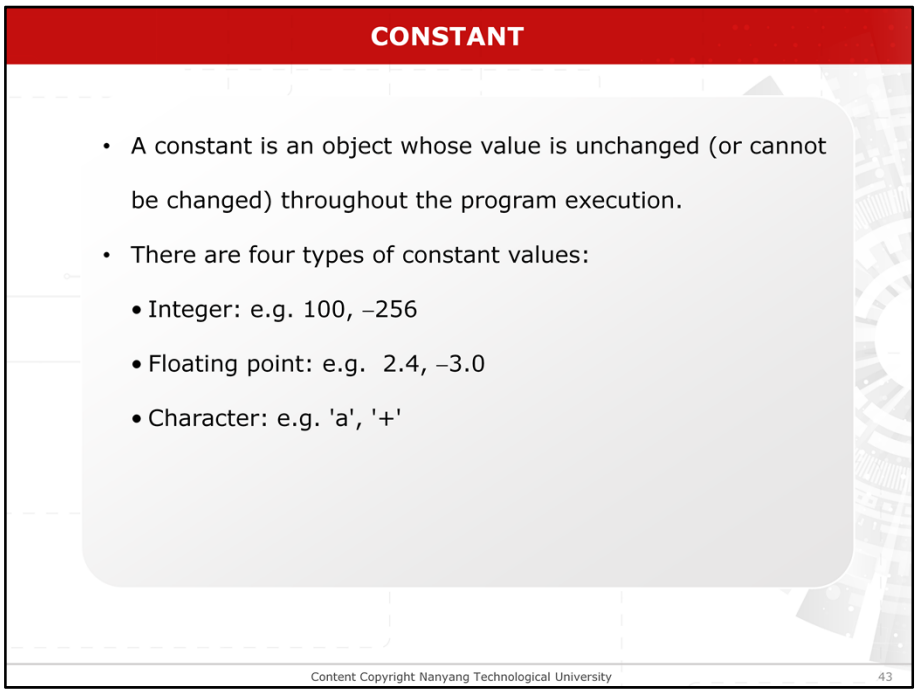
There are four types of constant values:



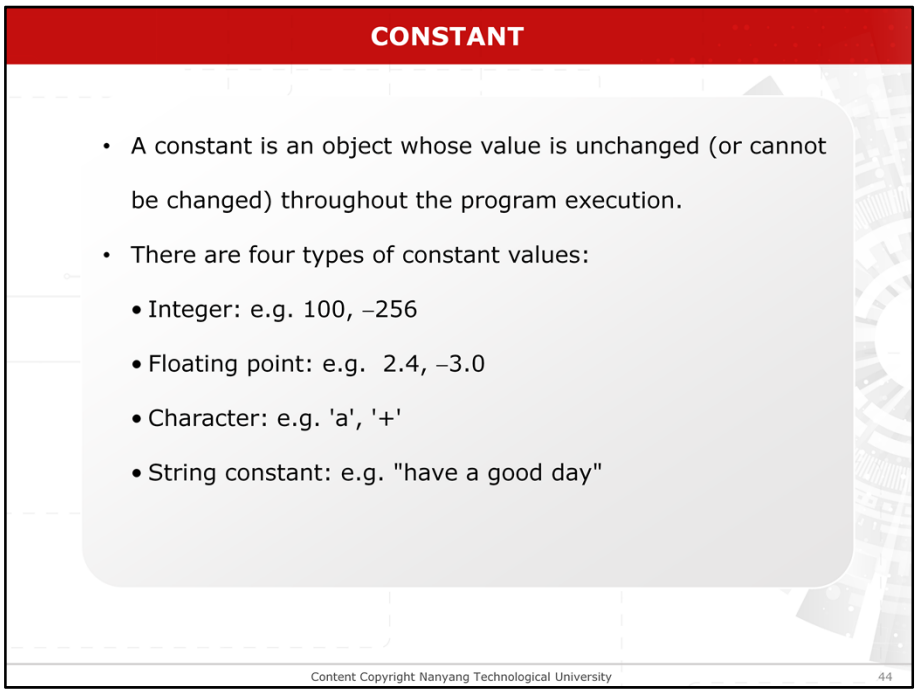
- Integer: e.g. 100, –256



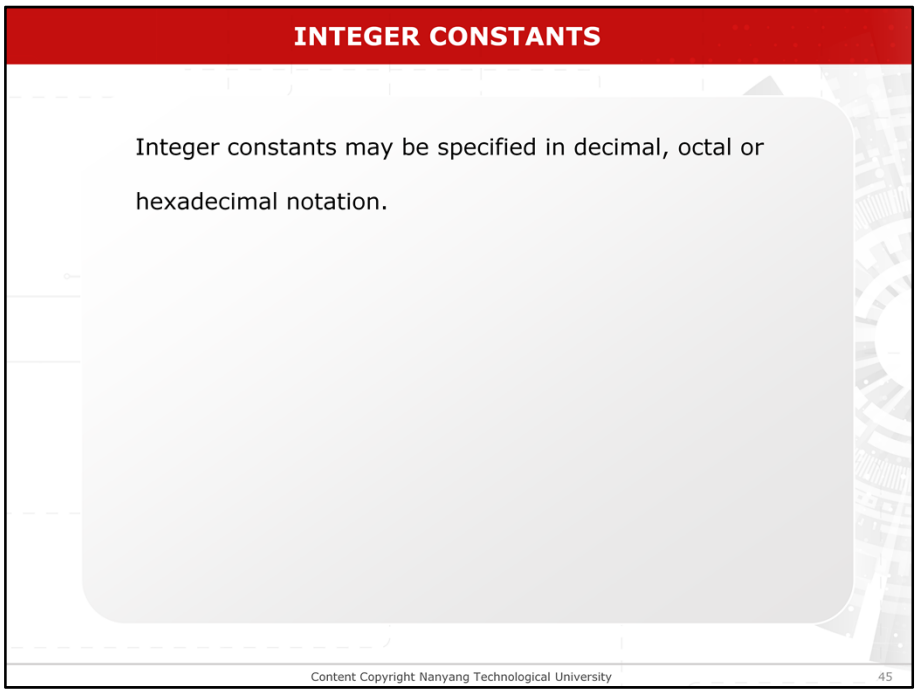
- floating point: e.g. 2.4, –3.0;



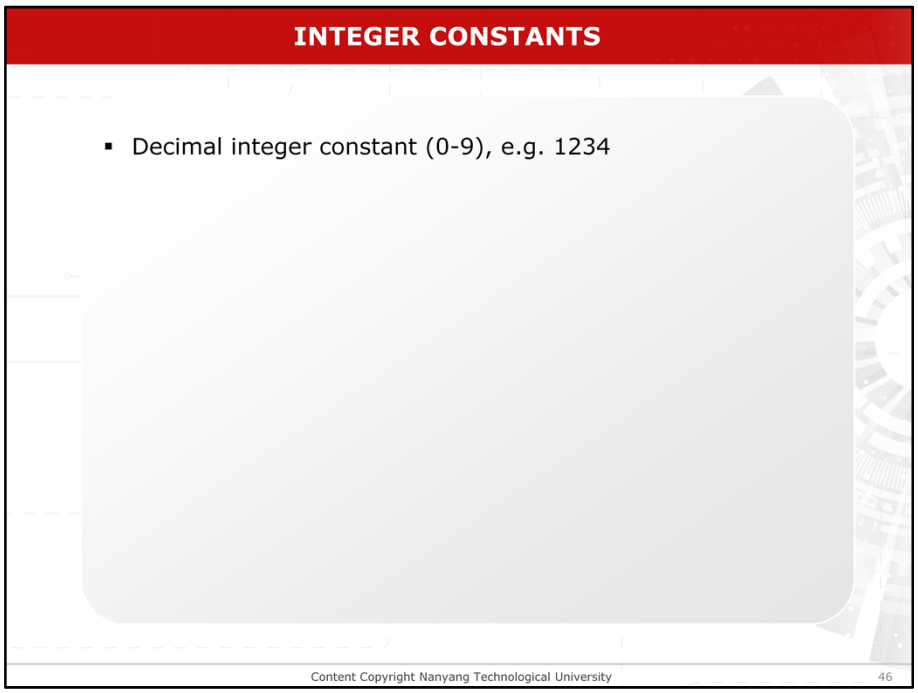
character: e.g. 'a', '+'



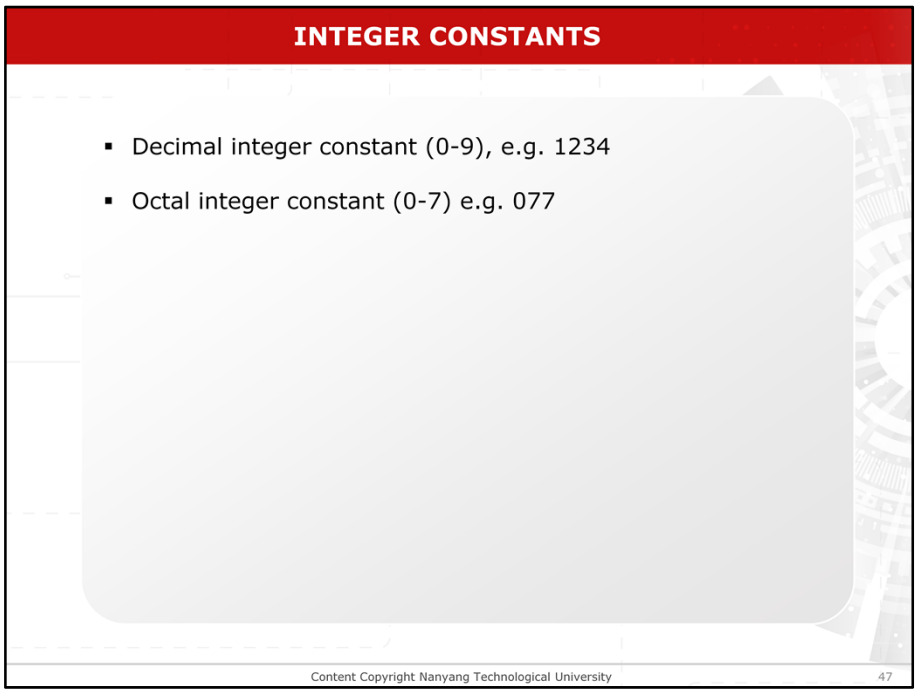
string constants which are a string of characters: e.g. "have a good day".



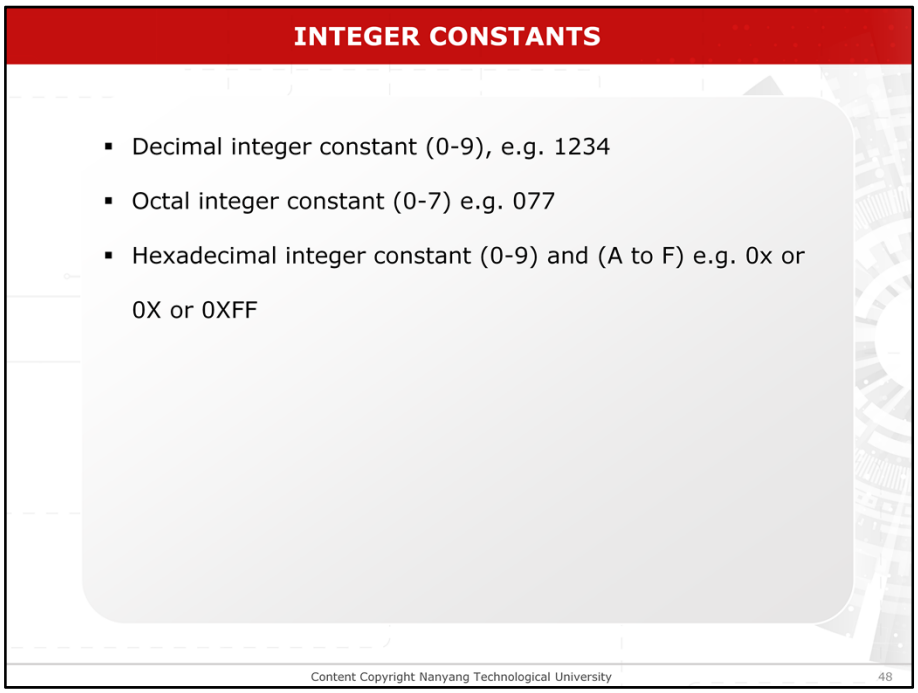
Integer constants may be specified in decimal, octal or hexadecimal notation.



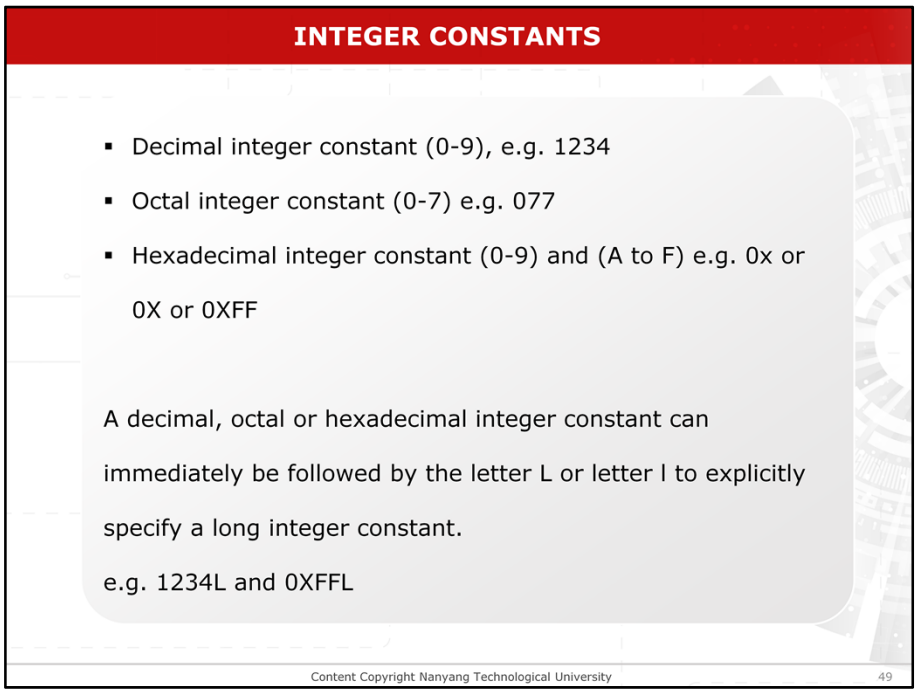
A decimal integer constant is written as a sequence of decimal digits (0-9), e.g. 1234.



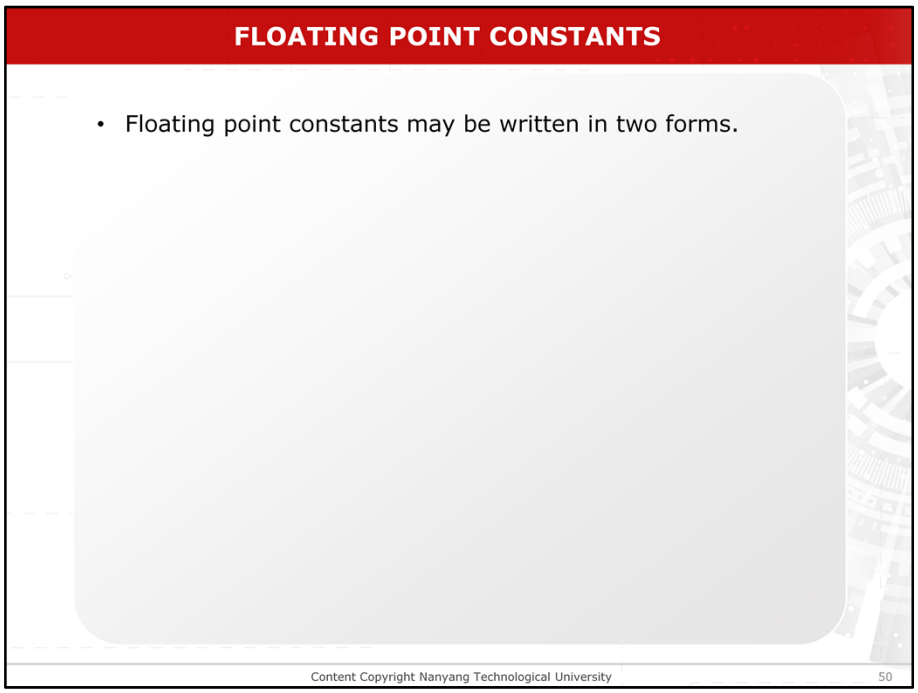
An octal integer constant is written as a sequence of octal digits (0-7) starting with a zero (**0**), e.g. **077**.



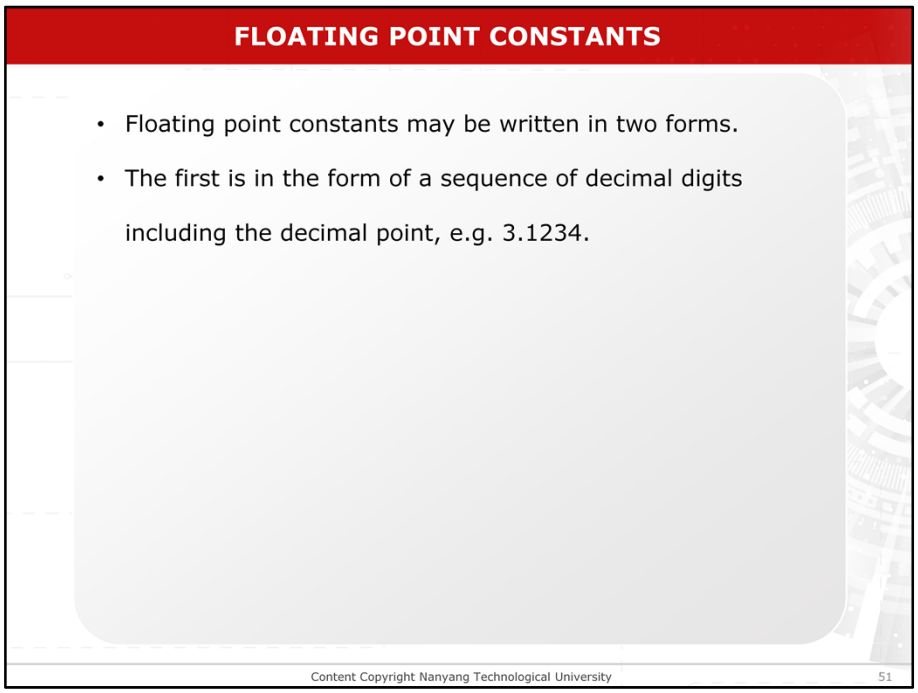
The hexadecimal digits comprise 0-9, and the letters A to F in uppercase or lowercase where the letters represent the values 10 to 15 respectively.



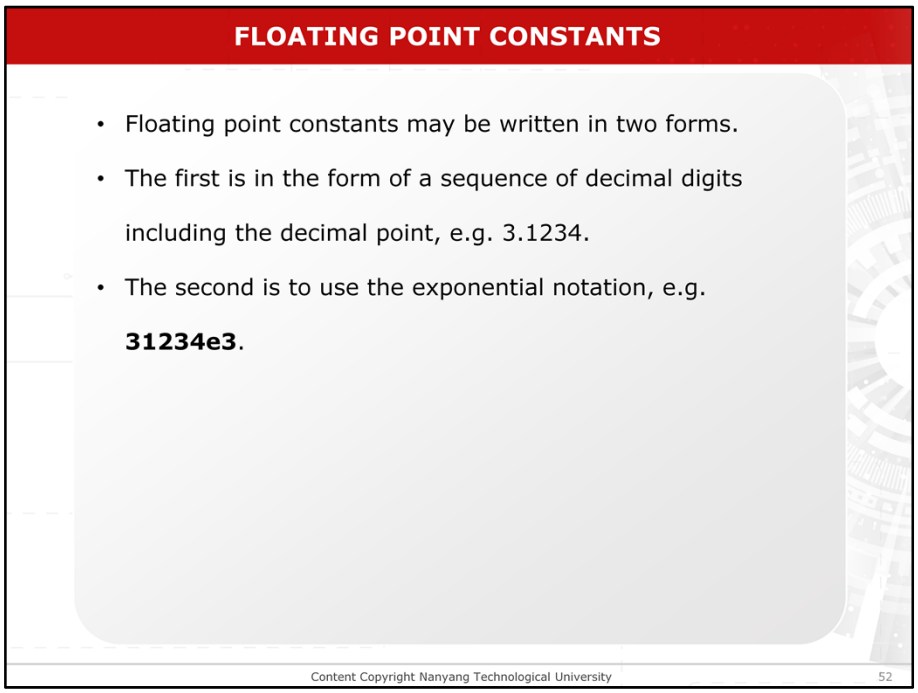
A decimal, octal or hexadecimal integer constant can immediately be followed by the letter L uppercase or lowercase to explicitly specify a long integer constant, e.g. 1234L and 0XFFL



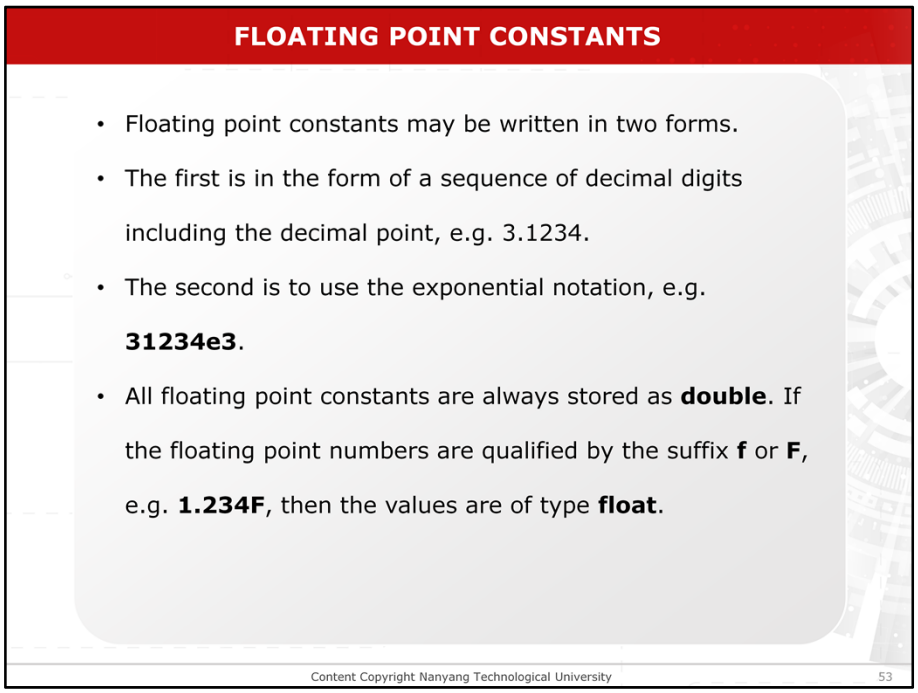
Floating point constants may be written in two forms.



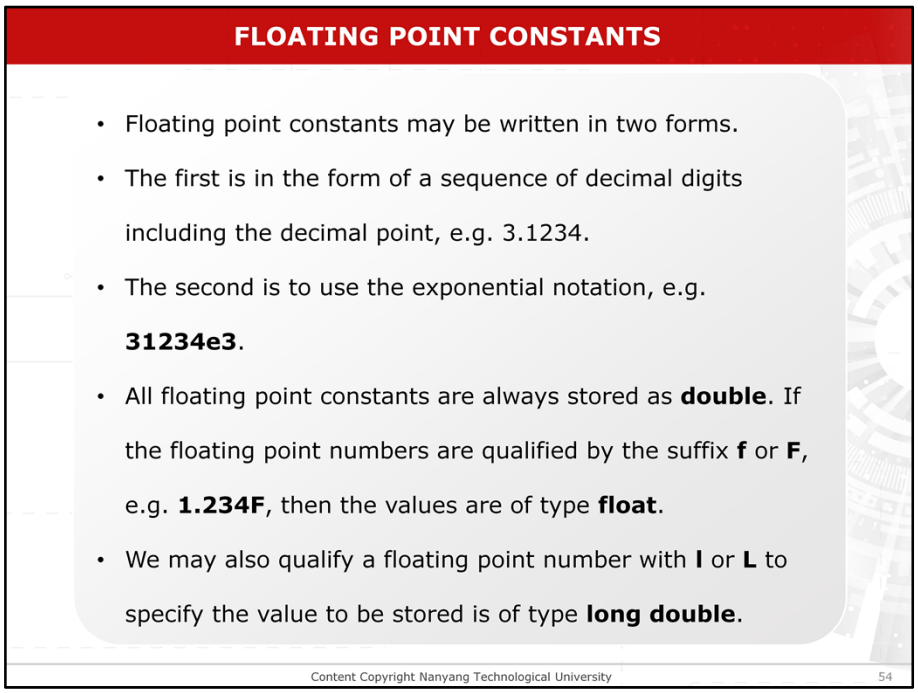
The first is in the form of a sequence of decimal digits including the decimal point, e.g. 3.1234.



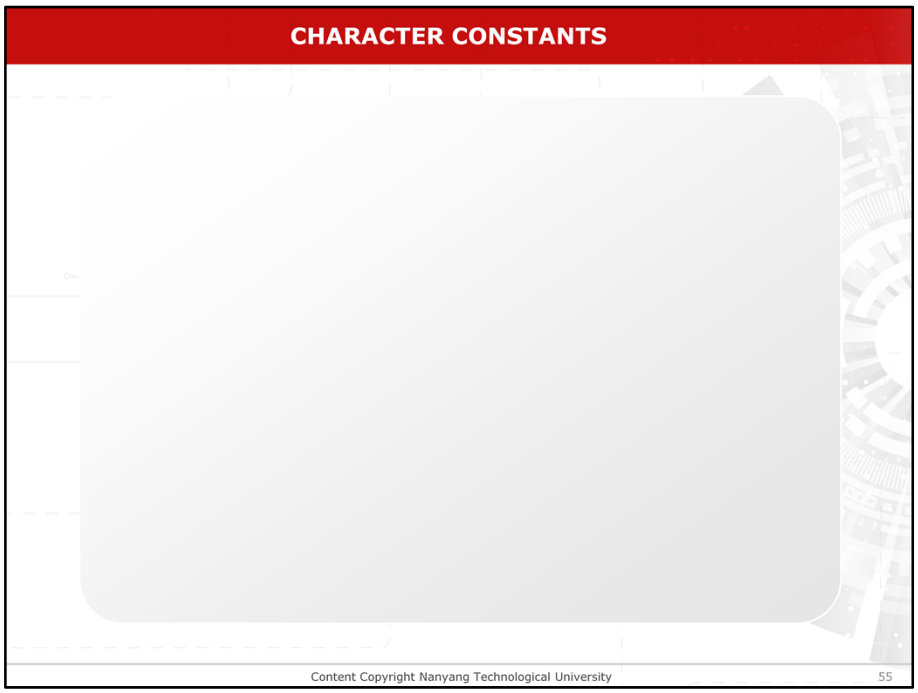
The second is to use the exponential notation as discussed in the previous section, e.g. **31234e3**.



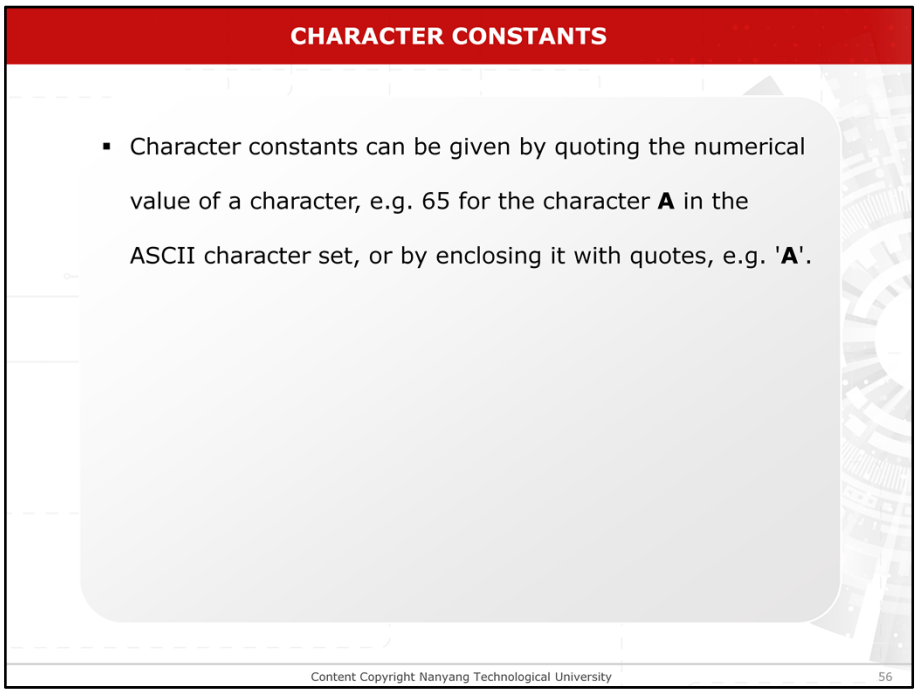
All floating point constants are always stored as double. If the floating point numbers are qualified by the suffix f LOWERCASE OR UPPERCASE e.g. 1.234F, then the values are of type float



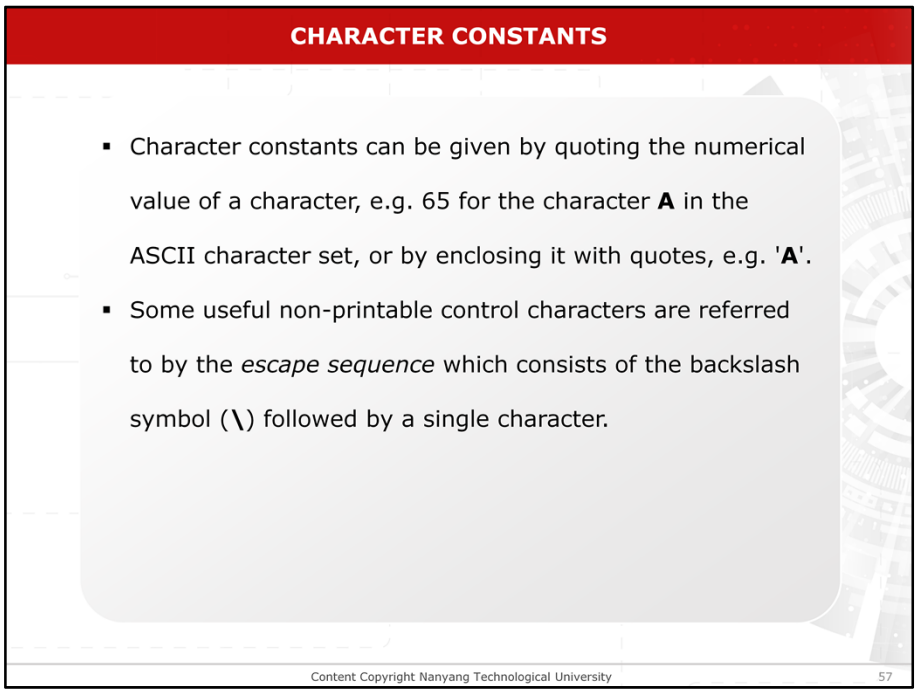
We may also qualify a floating point number with **l** or **L** to specify the value to be stored is of type **long double**.



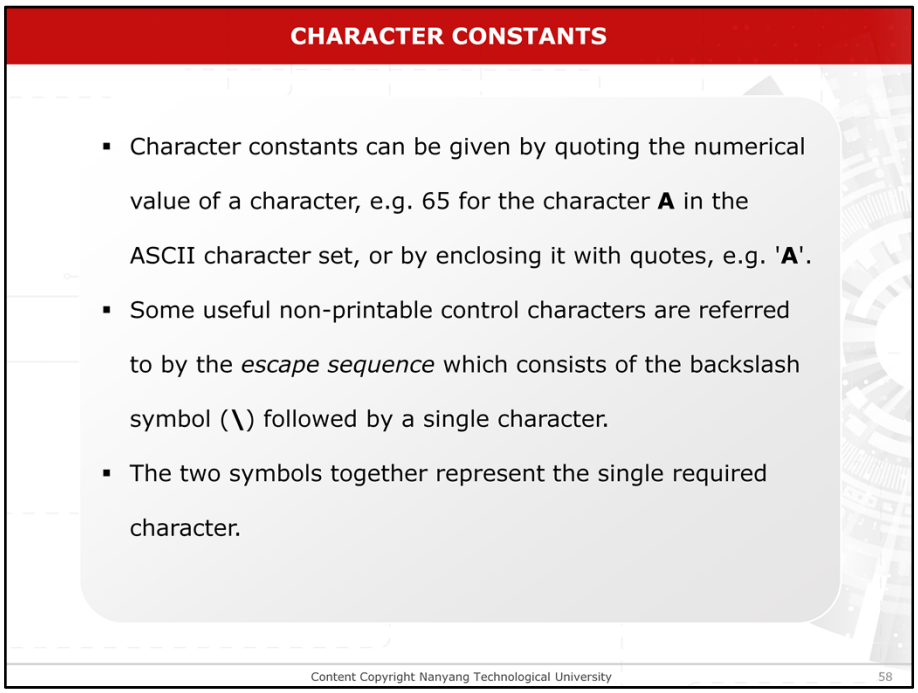
Character constants



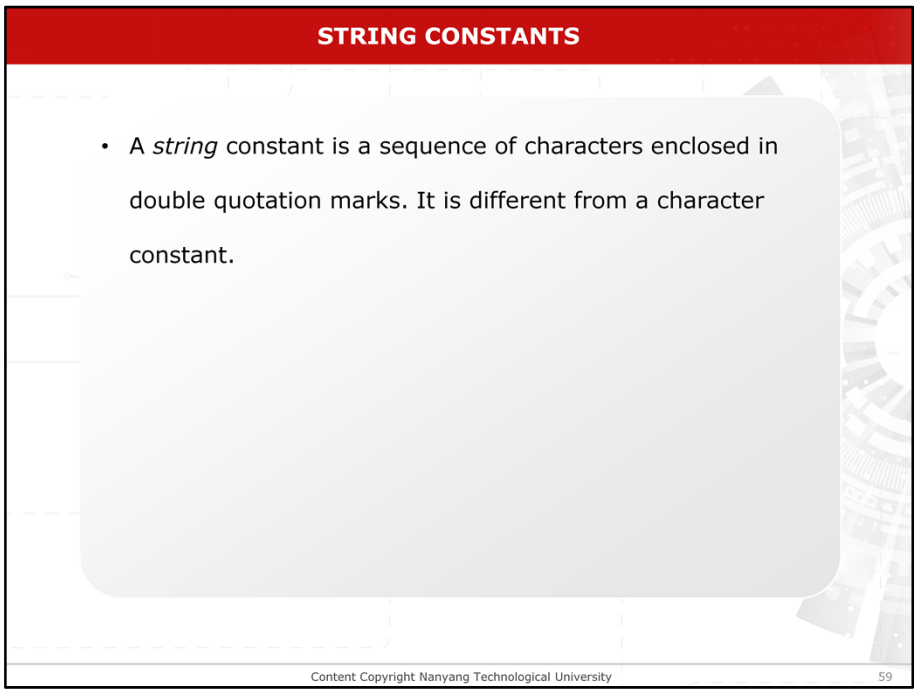
- Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **eh** in the ASCII character set, or by enclosing it with quotes



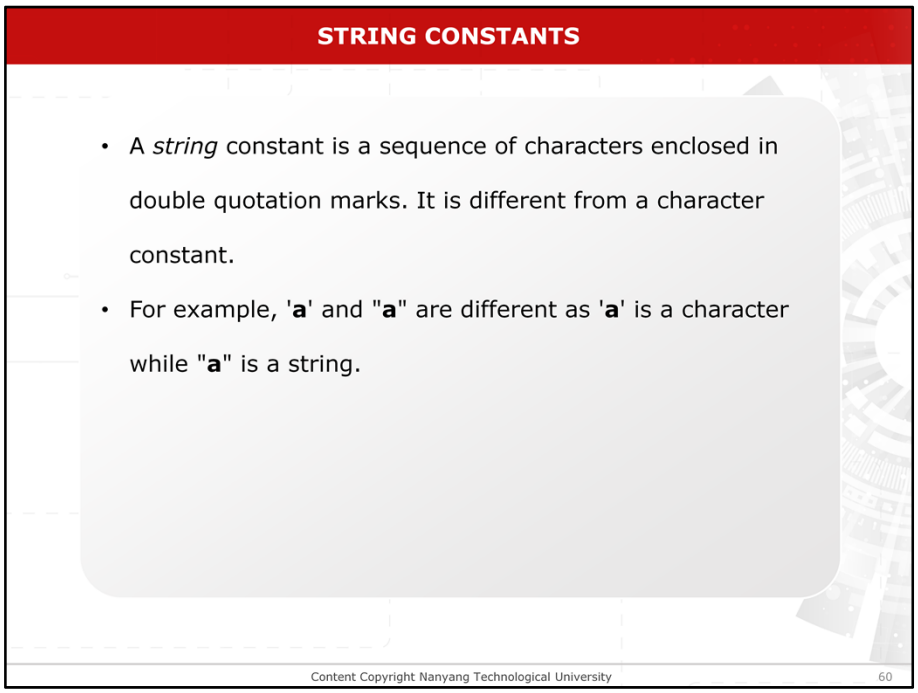
- Some useful non-printable control characters are referred to by the *escape sequence* which consists of the backslash symbol followed by a single character.



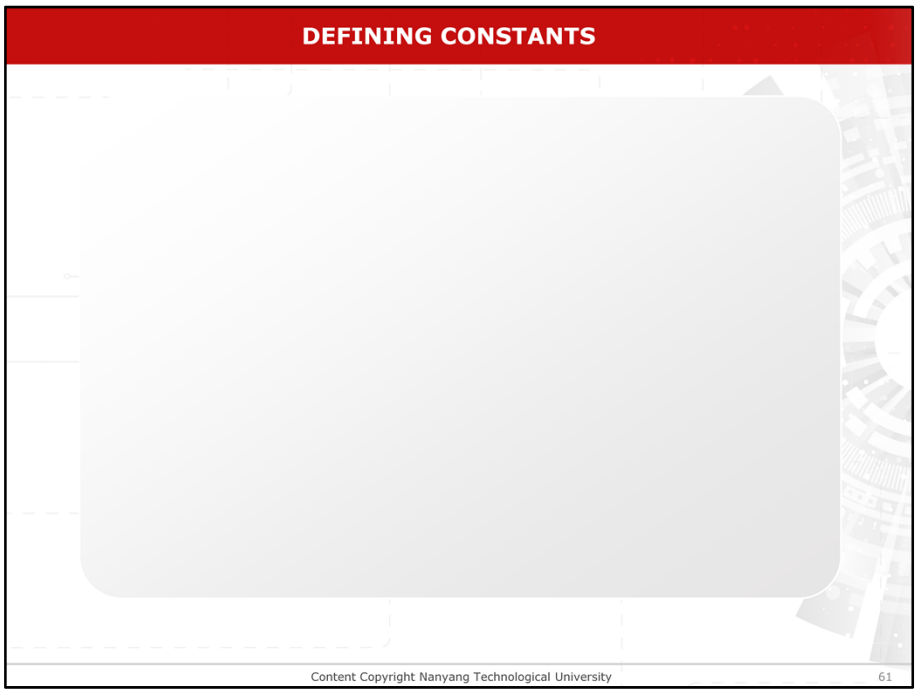
- The two symbols together represent the single required character. This is a better alternative, in terms of memorization, than ASCII numbers.



A *string* constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant.



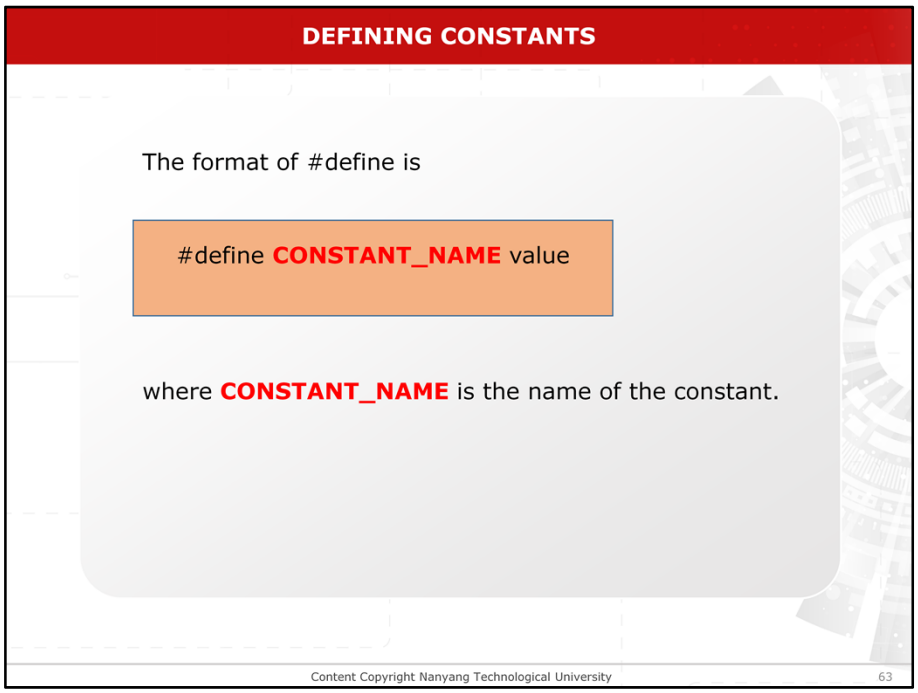
For example, ‘eh‘ in single quotes and “eh" in double quotes are different as ‘eh' in single quote is a character while “eh" in double quotes is a string. Strings will be discussed in details in the chapter on Character Strings



Defining Constants

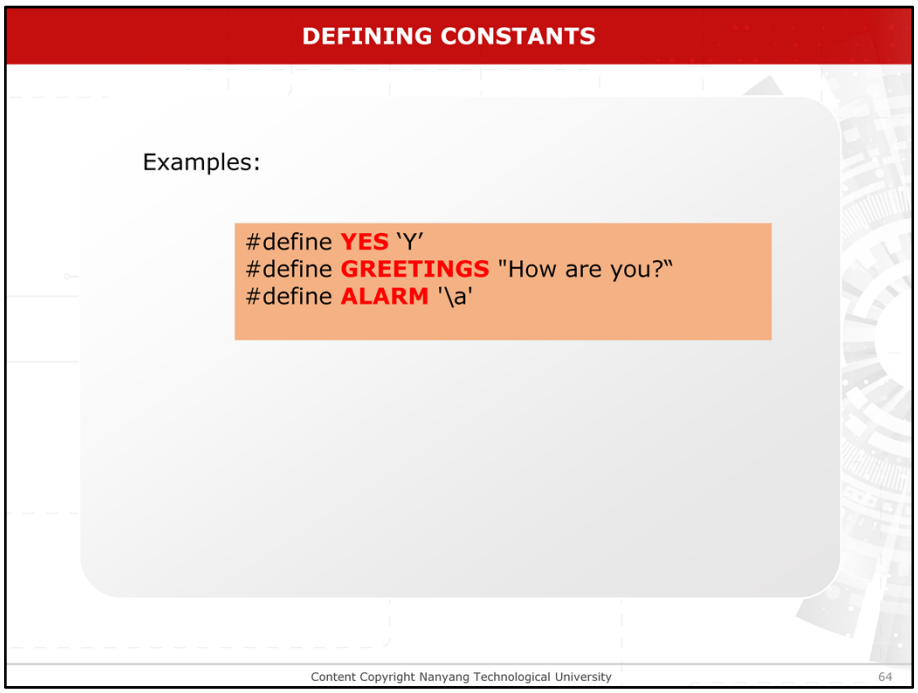
Defining Constants

We can use the preprocessor directive **#define as shown.**

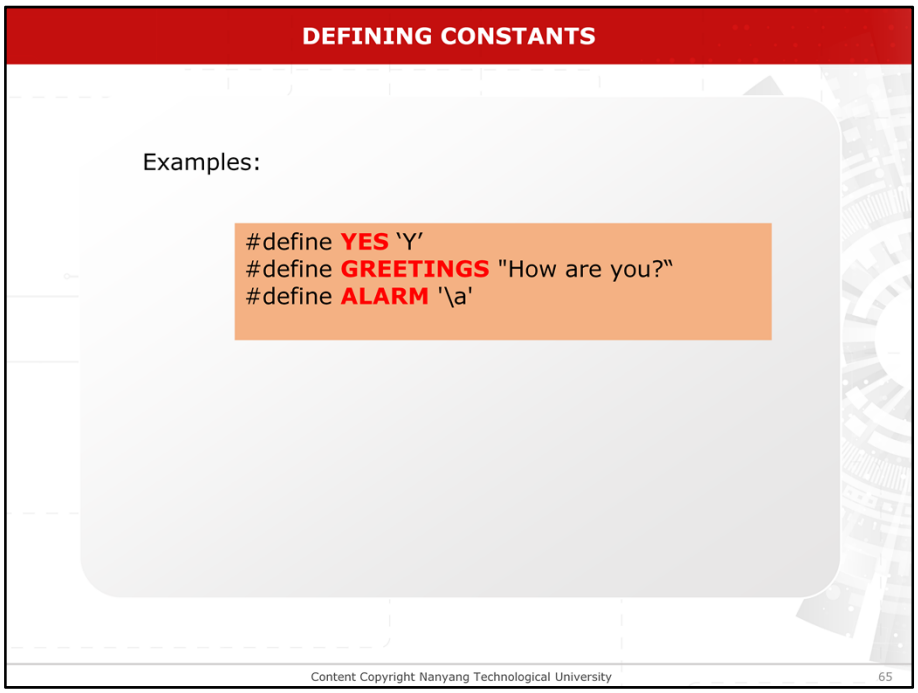


Defining Constants

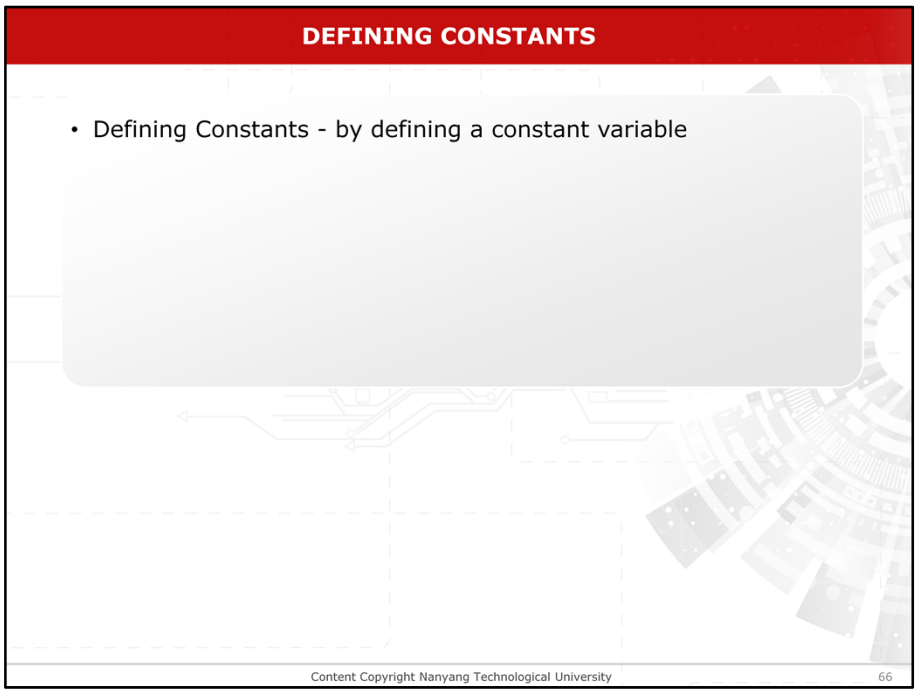
The format of **#define** is shown here., where **CONSTANT_NAME** is the name of the constant.



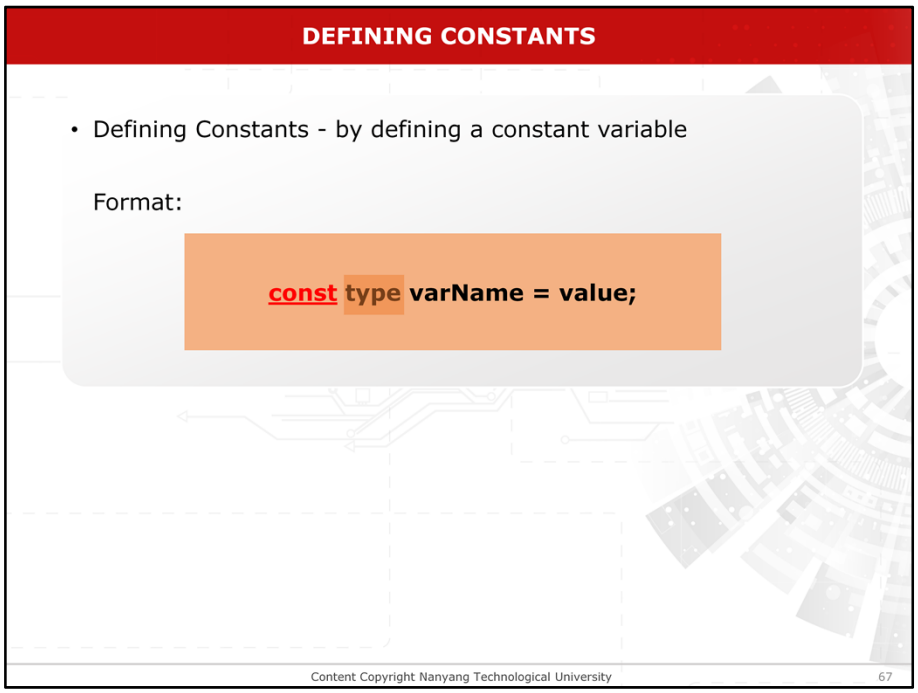
Some examples are shown here.



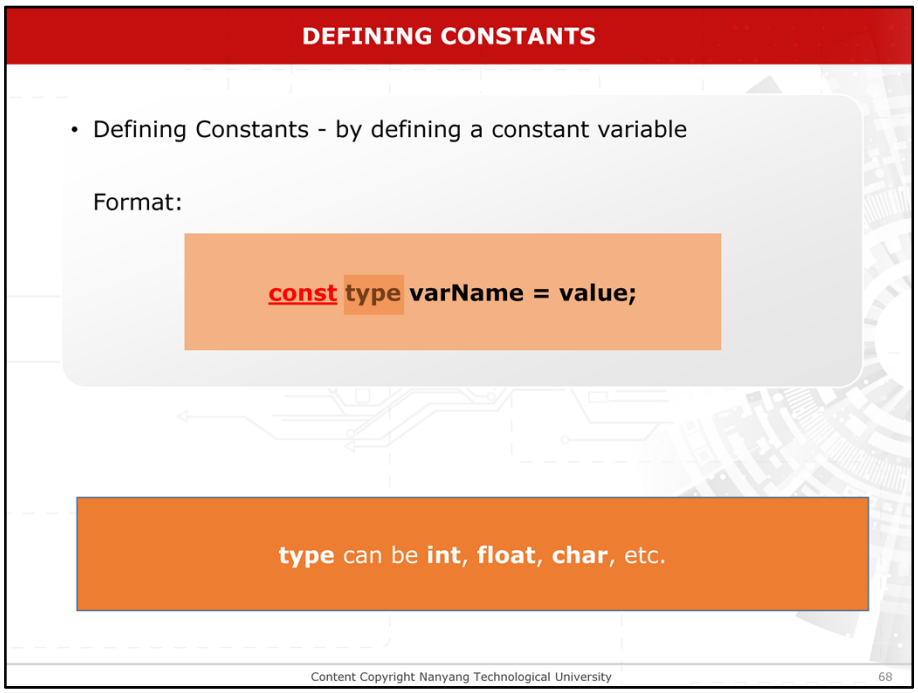
It is a C tradition that **CONSTANT_NAME** is in upper case. It makes programs more readable. During compilation, the value of the constant will substitute the name of the constant wherever it appears in the program. By giving a symbolic name to a constant, it improves the readability of the program and makes the program easier to be modified.



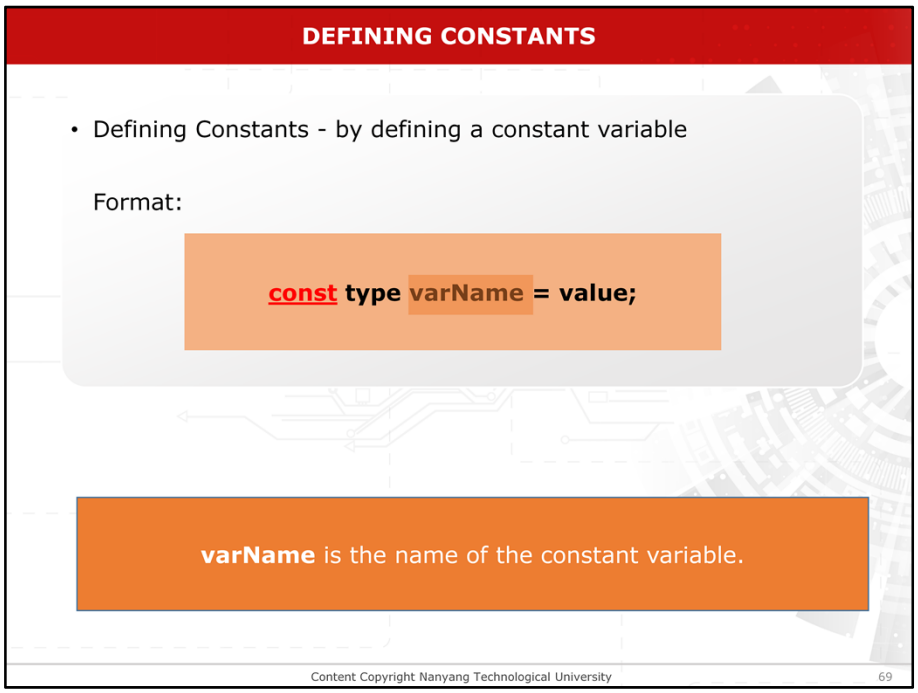
Another way to define a constant is to use a constant variable.



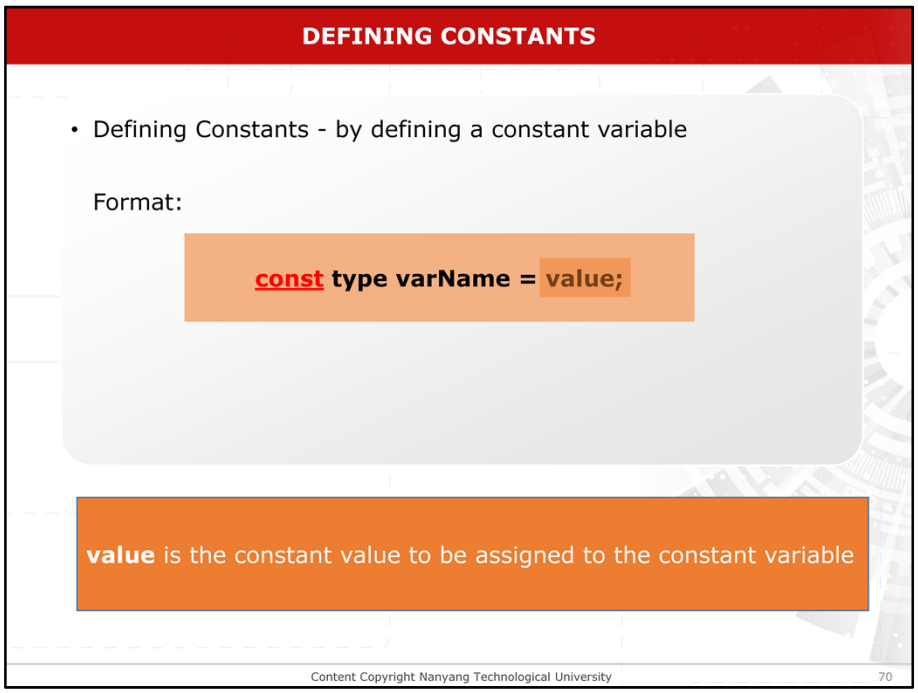
Format of defining constants is shown here.



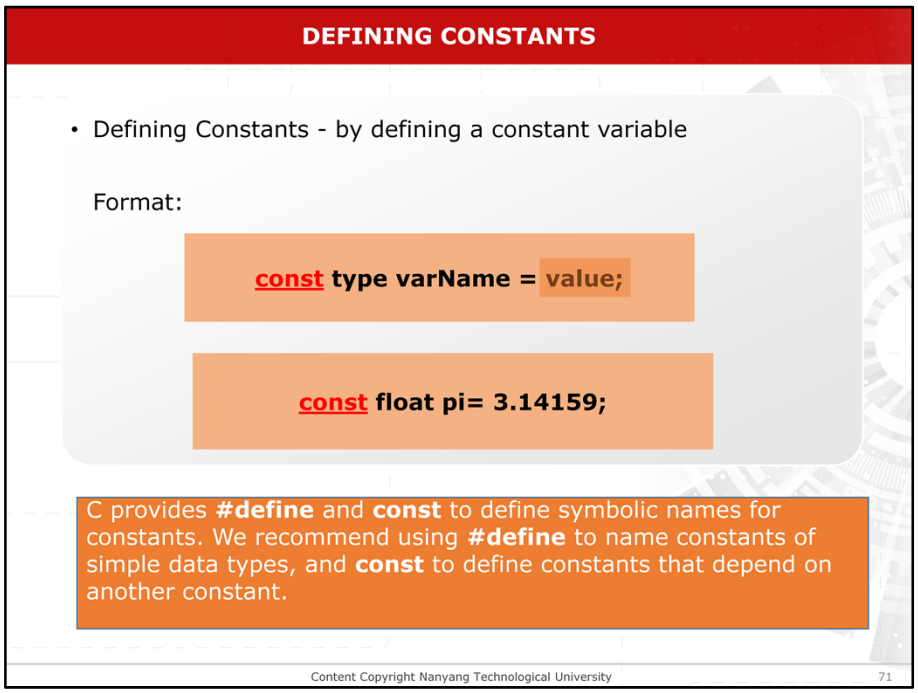
where **type** can be **int**, **float**, **car**, etc



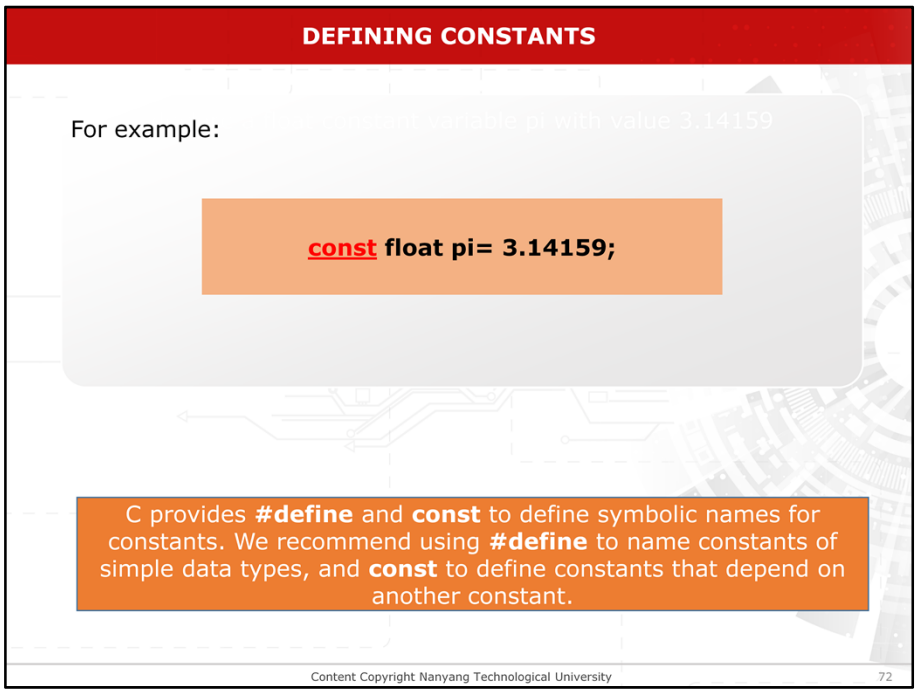
varName is the name of the constant variable



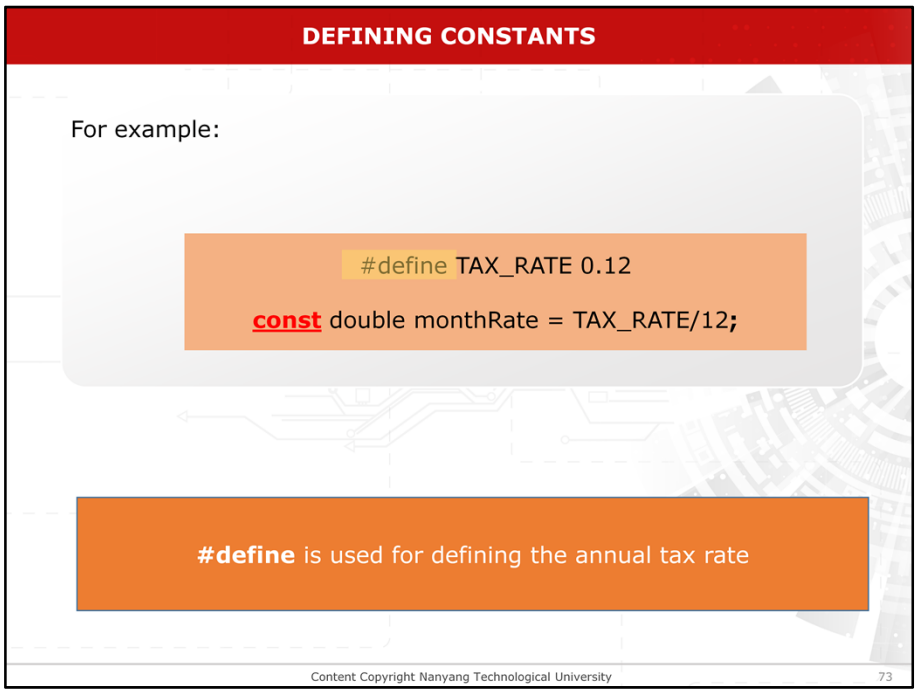
value is the constant value to be assigned to the constant variable



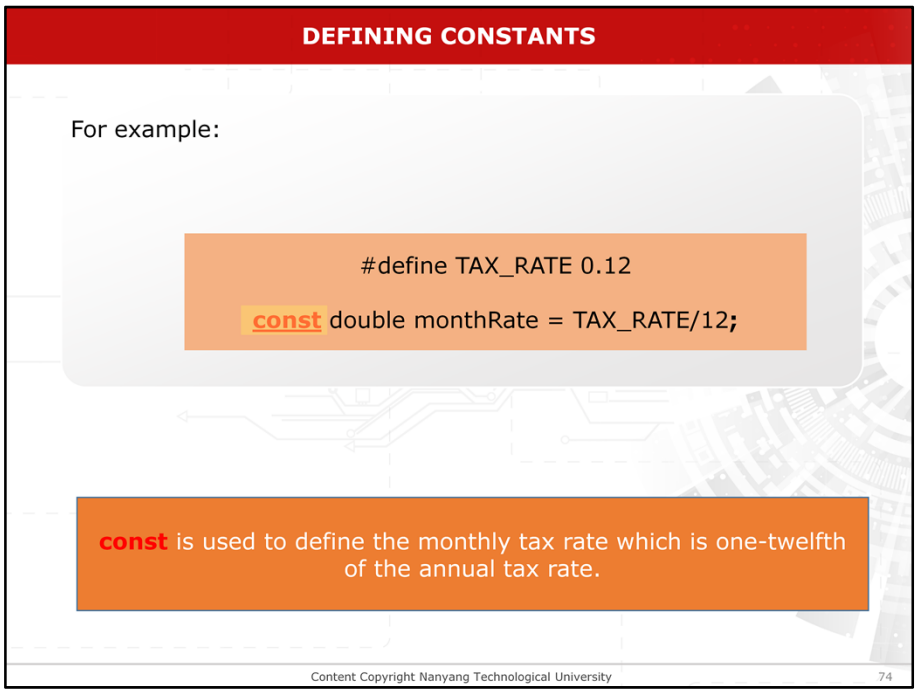
for example, const float pi = 3.14159; C provides #define and const to define symbolic names for constants



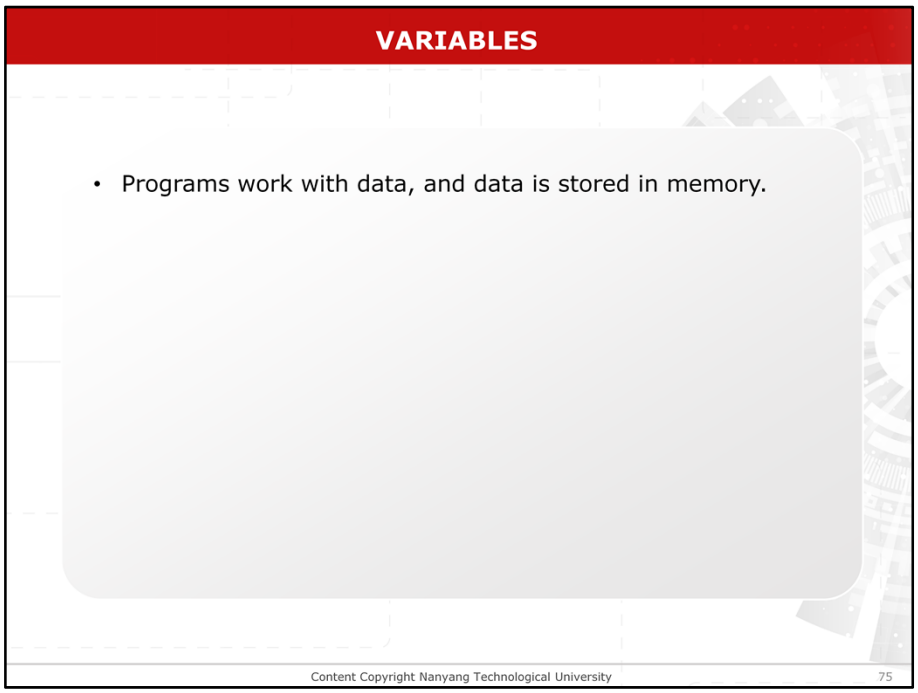
For example, it declares a float constant variable pi with value 3.14159.
C provides **#define** and **const** to define symbolic names for constants. We recommend using **#define** to name constants of simple data types, and **const** to define constants that depend on another constant.



For example, in the declarations
#define is used for defining the annual tax rate

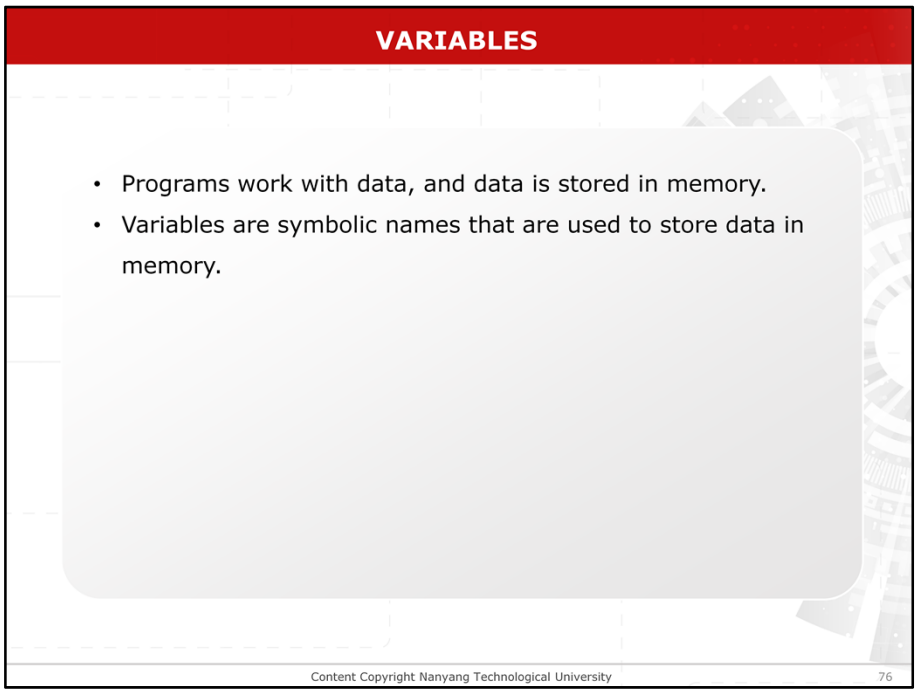


While const is used to define the monthly tax rate which is one-twelfth of the annual tax rate.

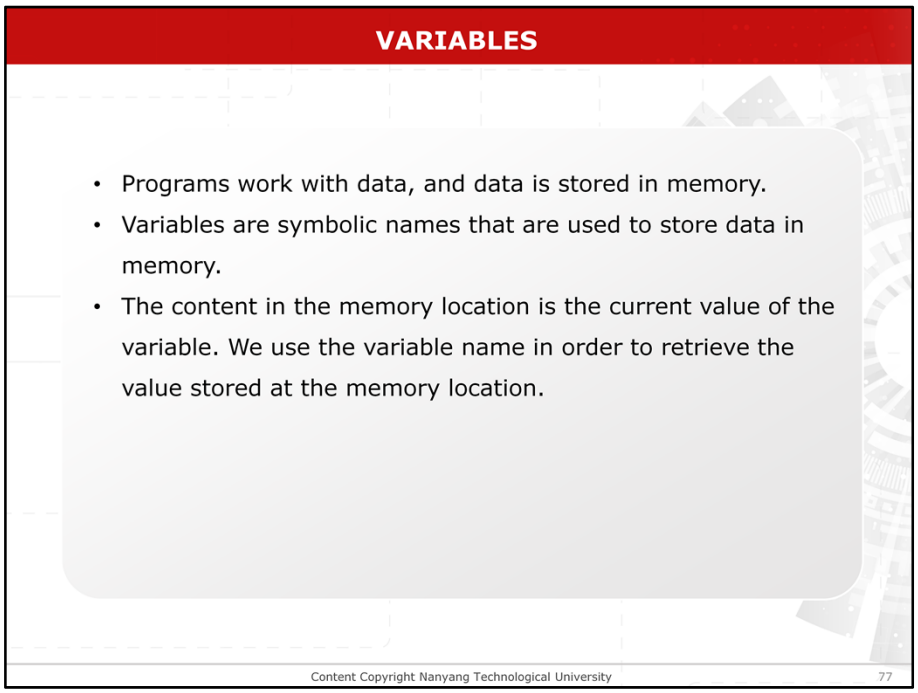


Variables

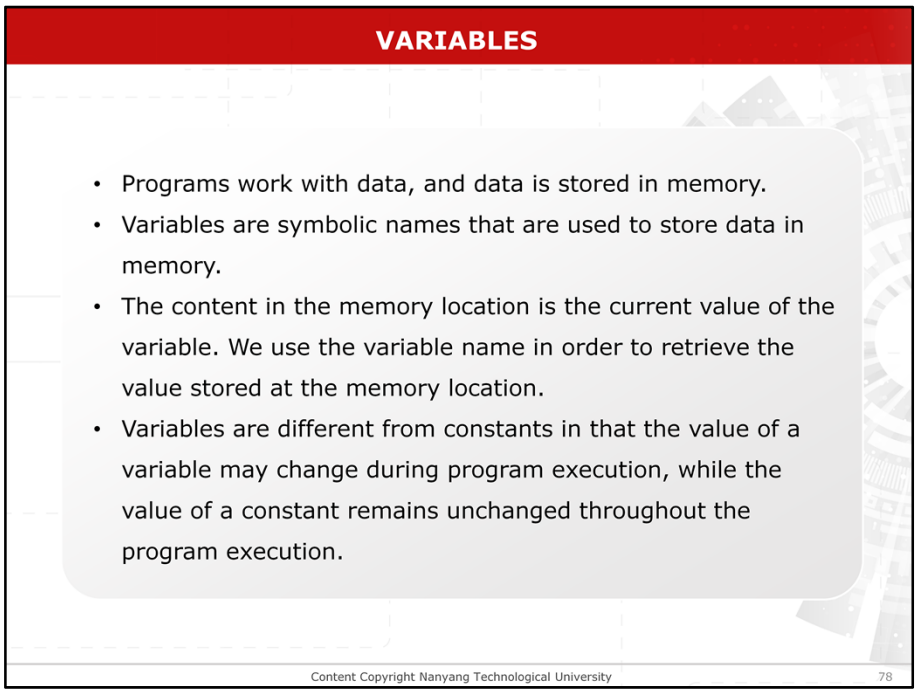
Programs work with data, and data is stored in memory.



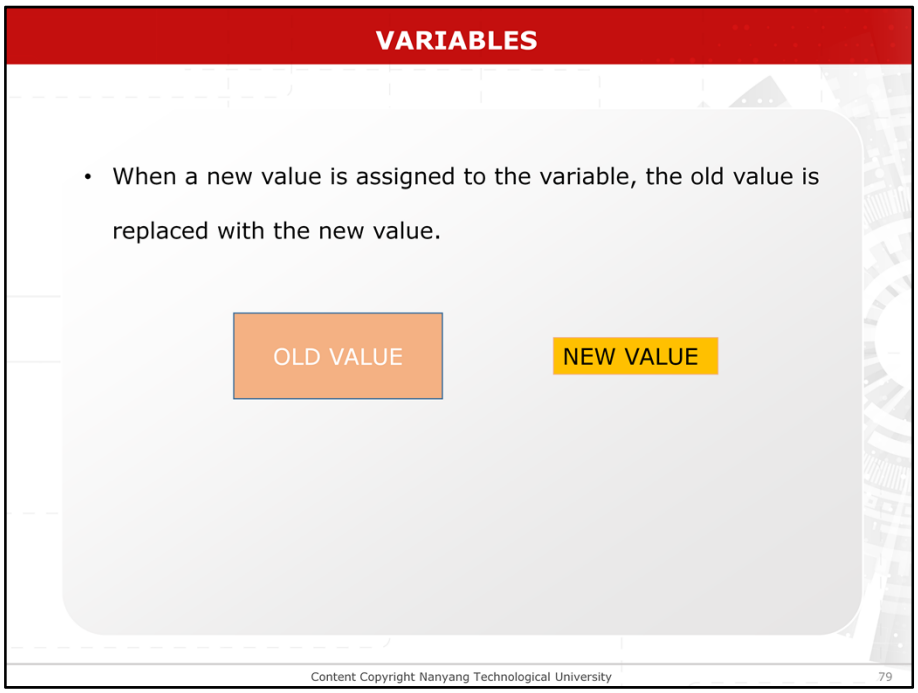
Variables are symbolic names that are used to store data in memory.



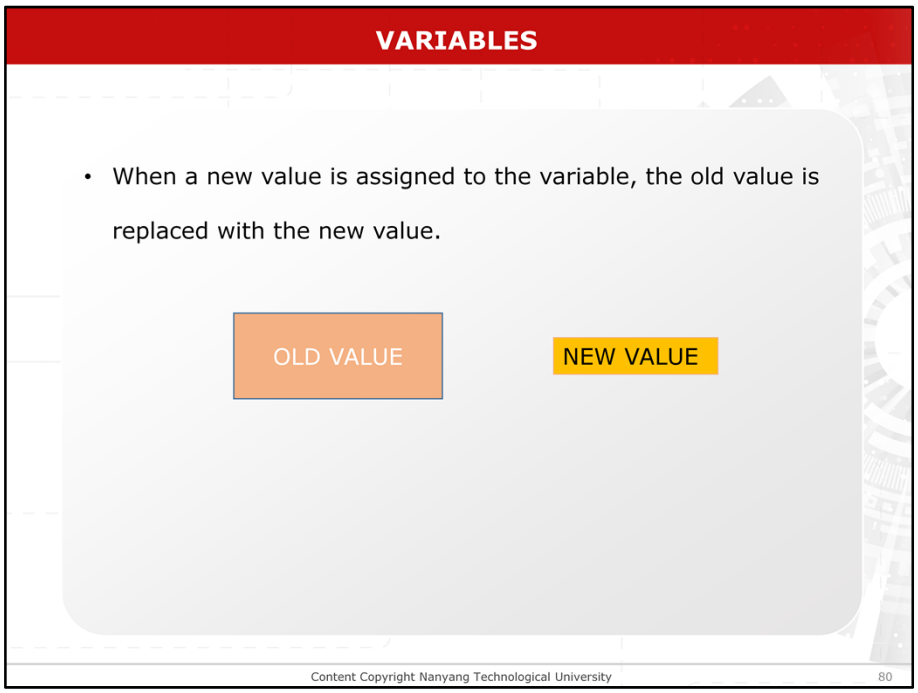
The content in the memory location is the current value of the variable. We use the variable name in order to retrieve the value stored at the memory location.



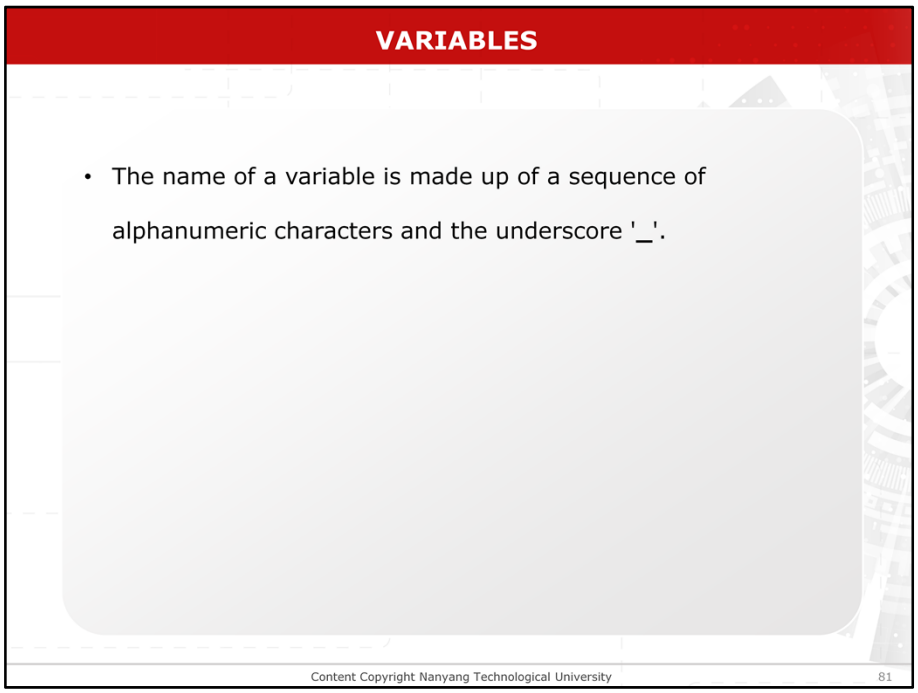
Variables are different from constants in that the value of a variable may change during program execution, while the value of a constant remains unchanged throughout the program execution.



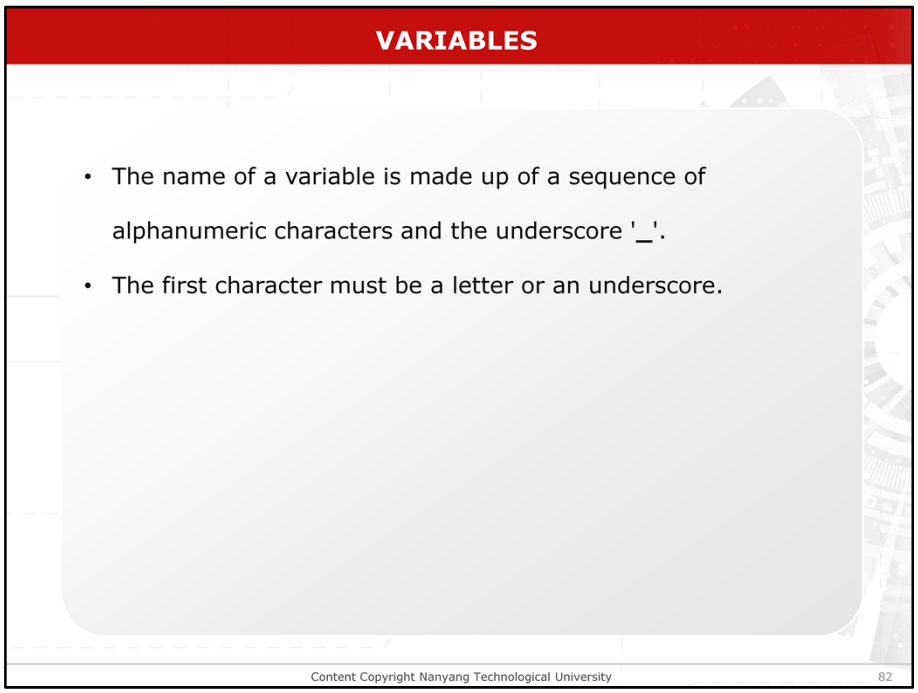
When a new value is assigned to the variable, the old value is replaced with the new value.



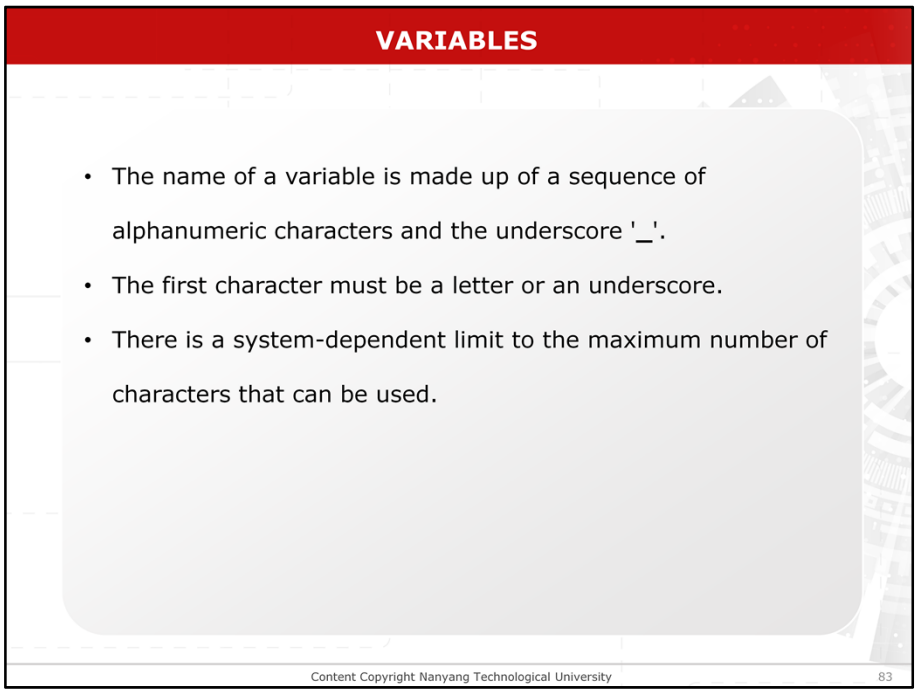
When a new value is assigned to the variable, the old value is replaced with the new value.



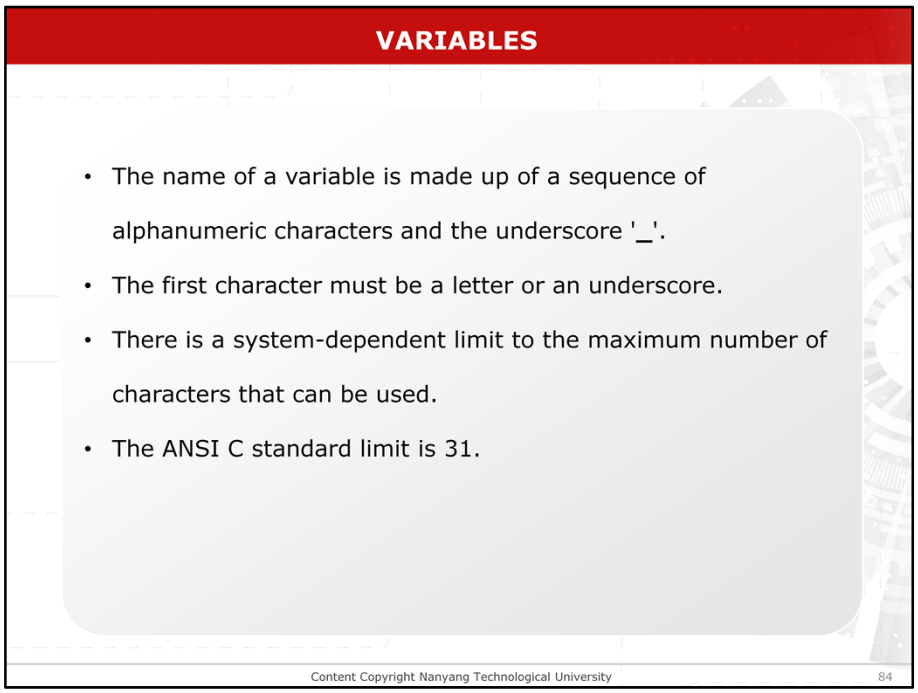
The name of a variable is made up of a sequence of alphanumeric characters and the underscore.



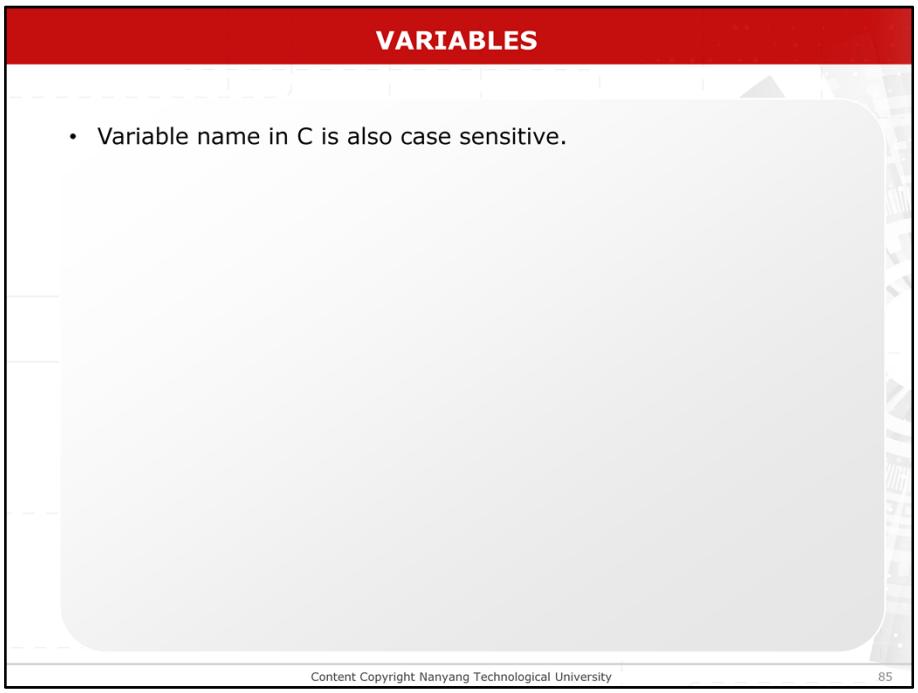
The first character must be a letter or an underscore.



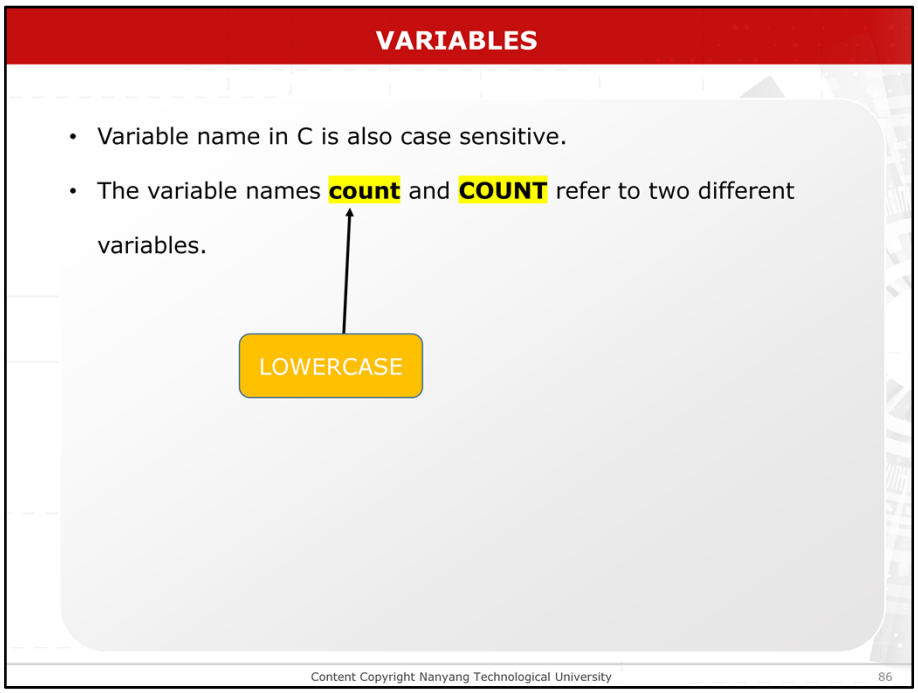
There is a system-dependent limit to the maximum number of characters that can be used.



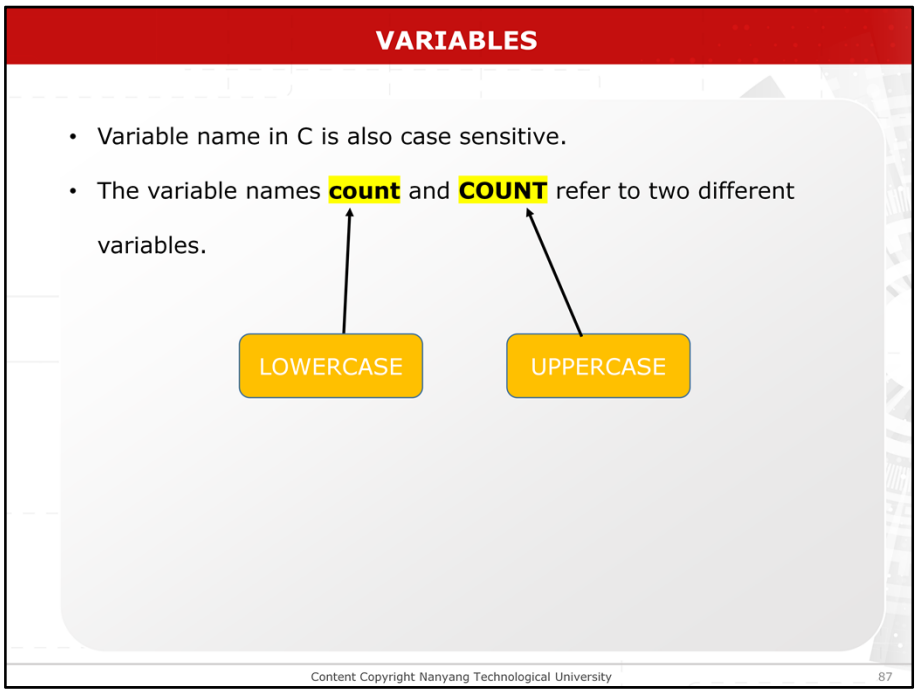
The ANSI C standard limit is 31.



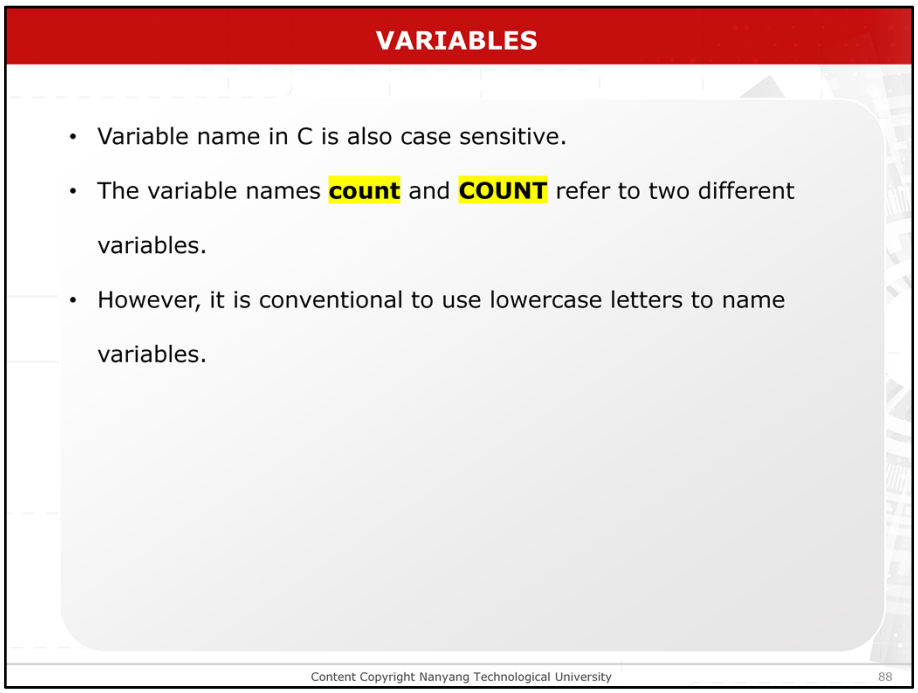
Variable name in C is also case sensitive.



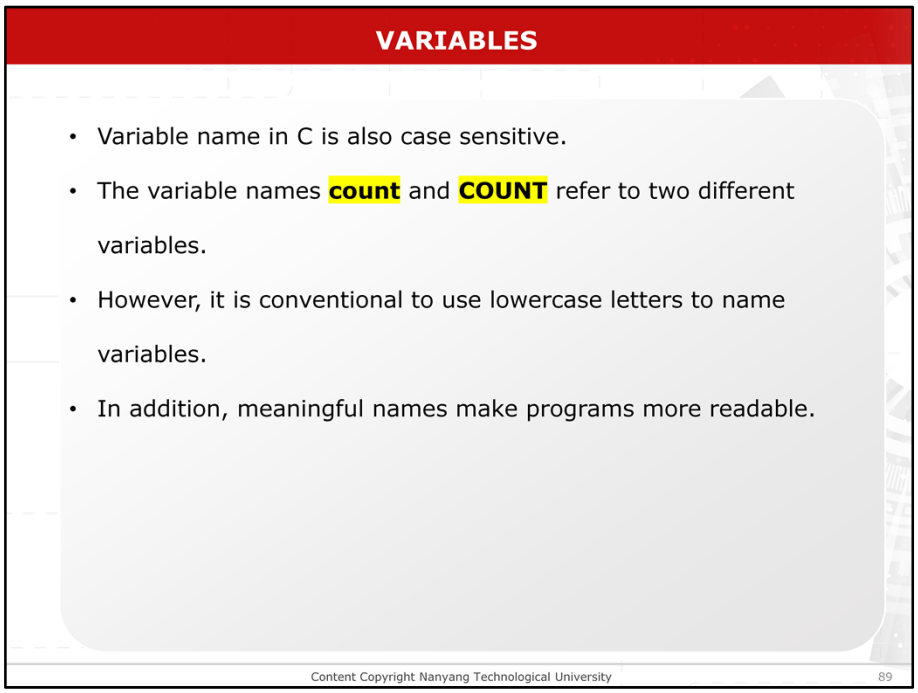
The variable names **count WRITTEN IN LOWER CASE**



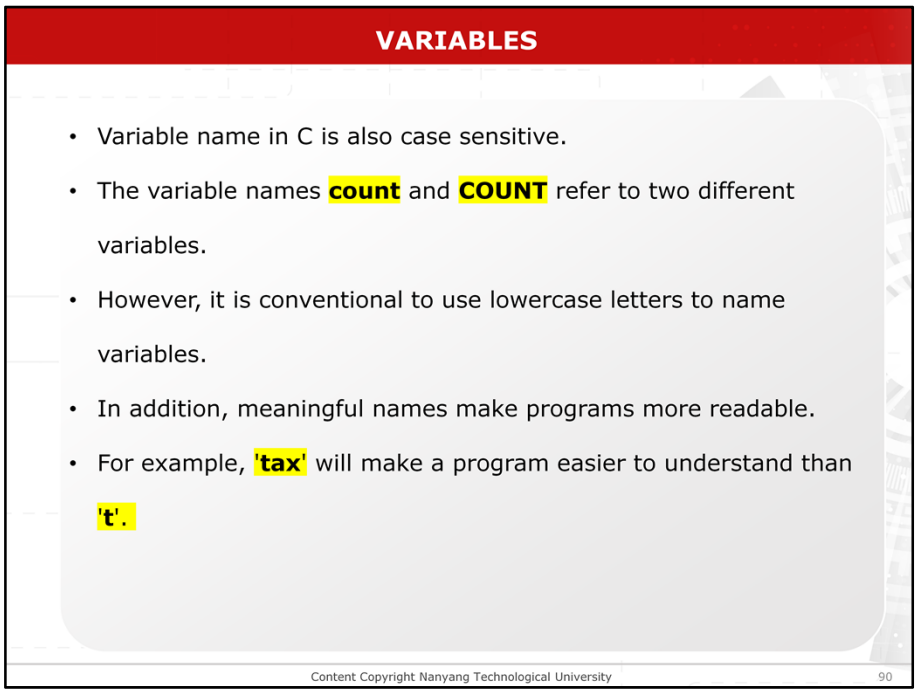
COUNT written in uppercase refer to two different variables.



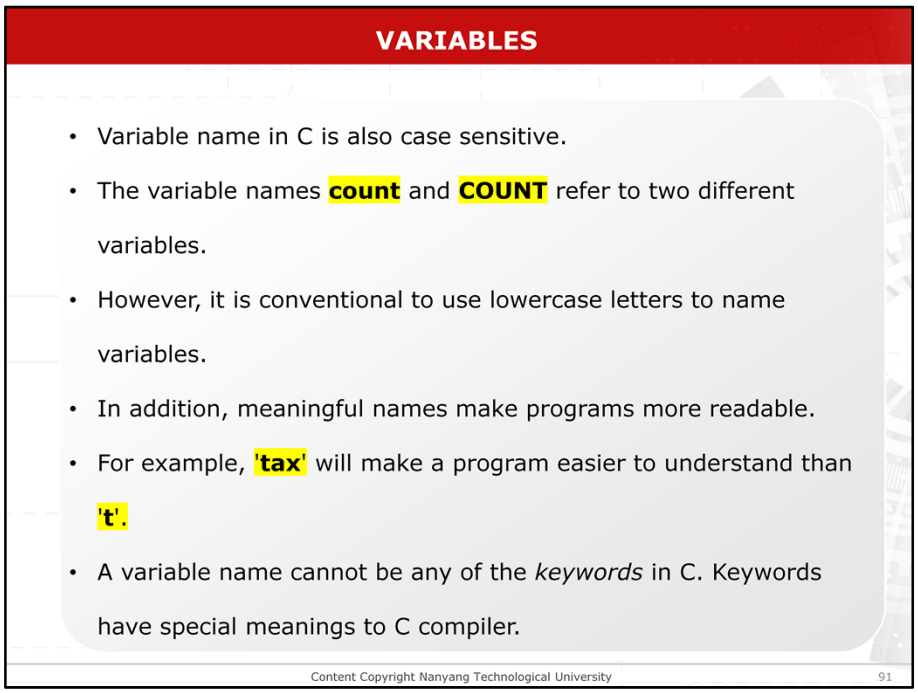
However, it is conventional to use lowercase letters to name variables.



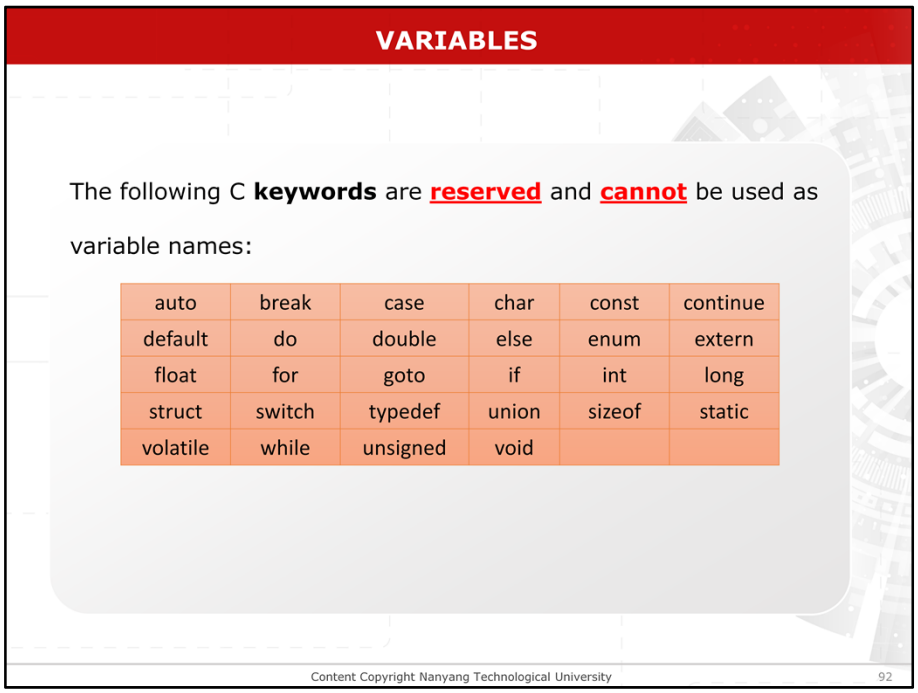
In addition, meaningful names make programs more readable.



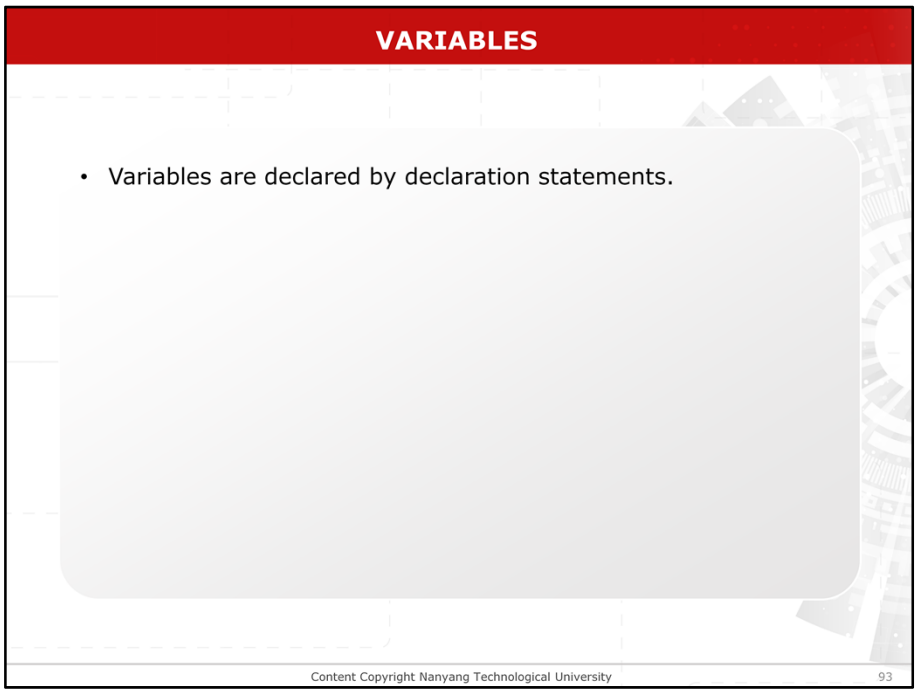
For example, '**tax**' will make a program easier to understand than '**t**'.



A variable name cannot be any of the *keywords* in C. ()() Keywords have special meanings to C compiler.



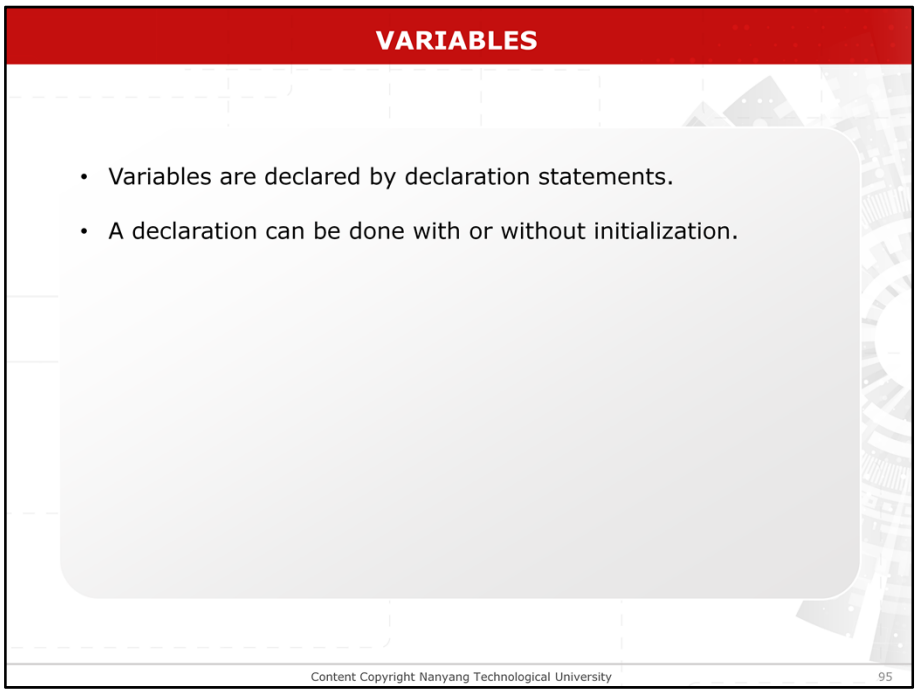
The following C **keywords** are **reserved** and **cannot** be used as variable names:



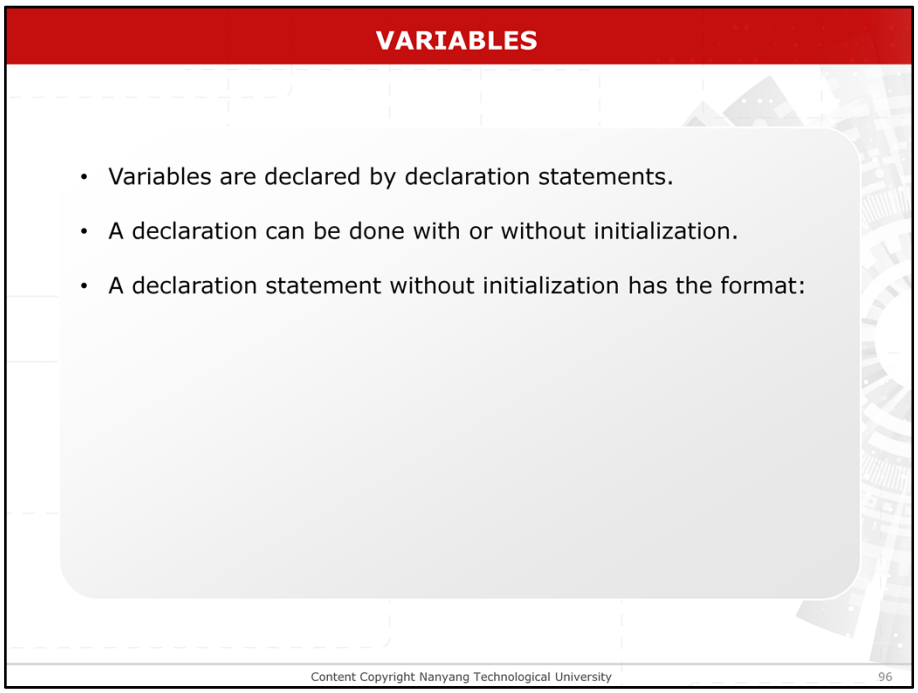
Each variable has a type. The basic C data types are integer, floating point and character. Variables are declared by *declaration statements.*

VARIABLES

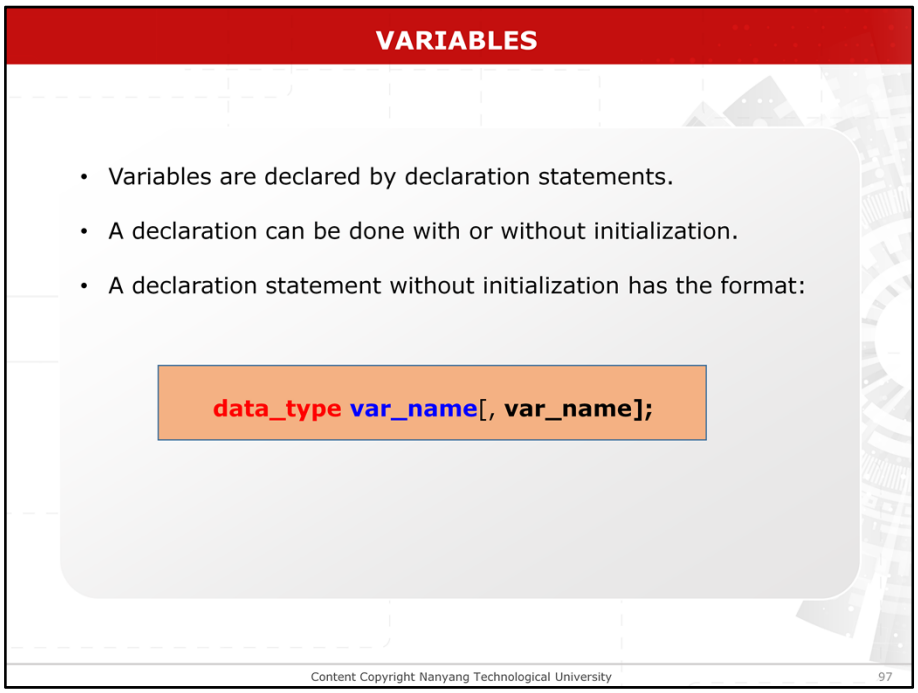
- Variables are declared by declaration statements.



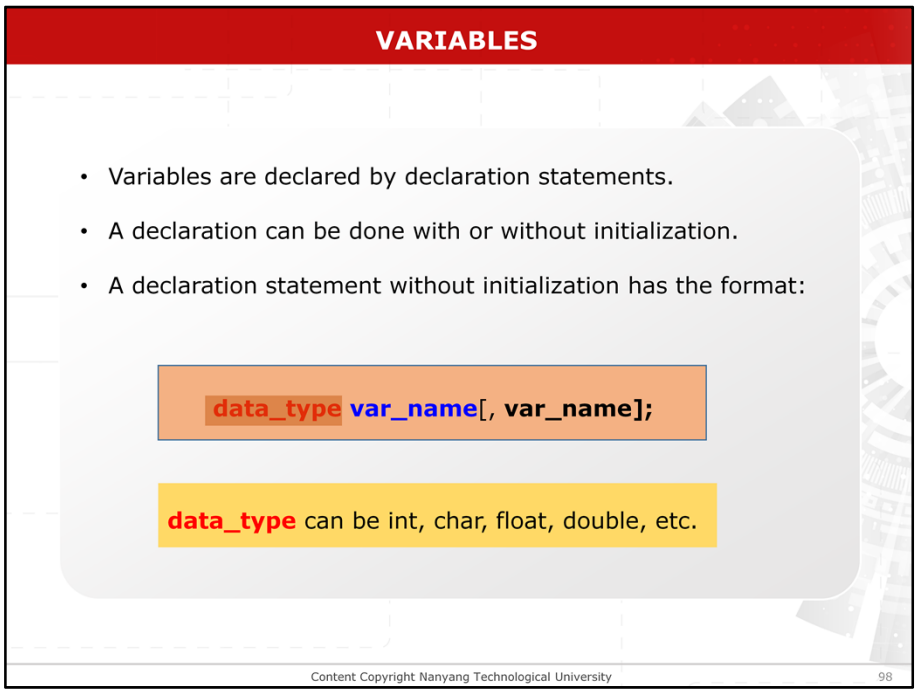
A declaration can be done with or without initialization.



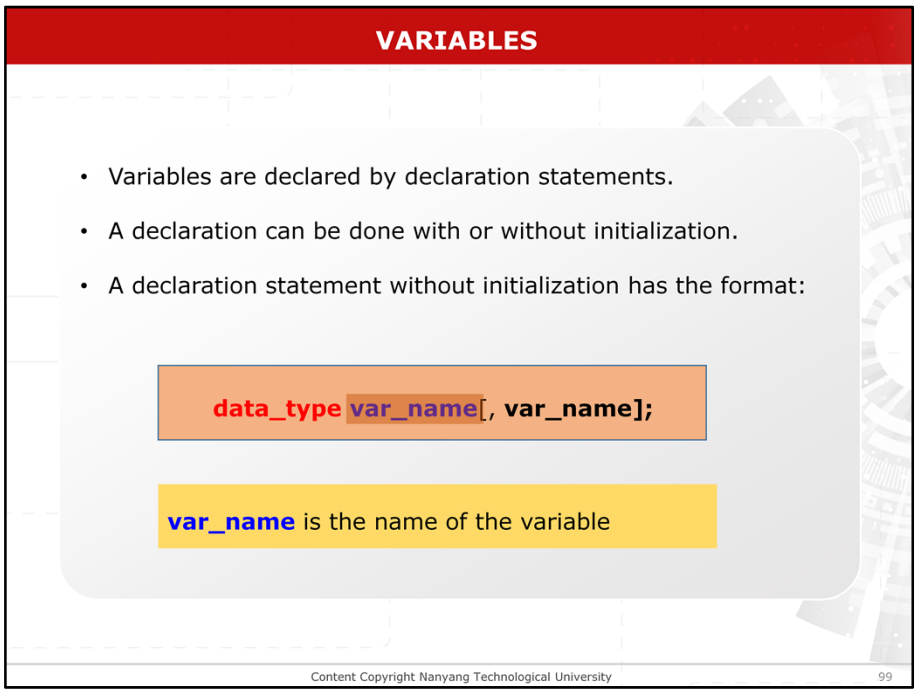
A declaration statement without initialization has the format


VARIABLES
• Variables are declared by declaration statements.
• A declaration can be done with or without initialization.
• A declaration statement without initialization has the format:
data_type var_name[, var_name];
Content Copyright Nanyang Technological University
97

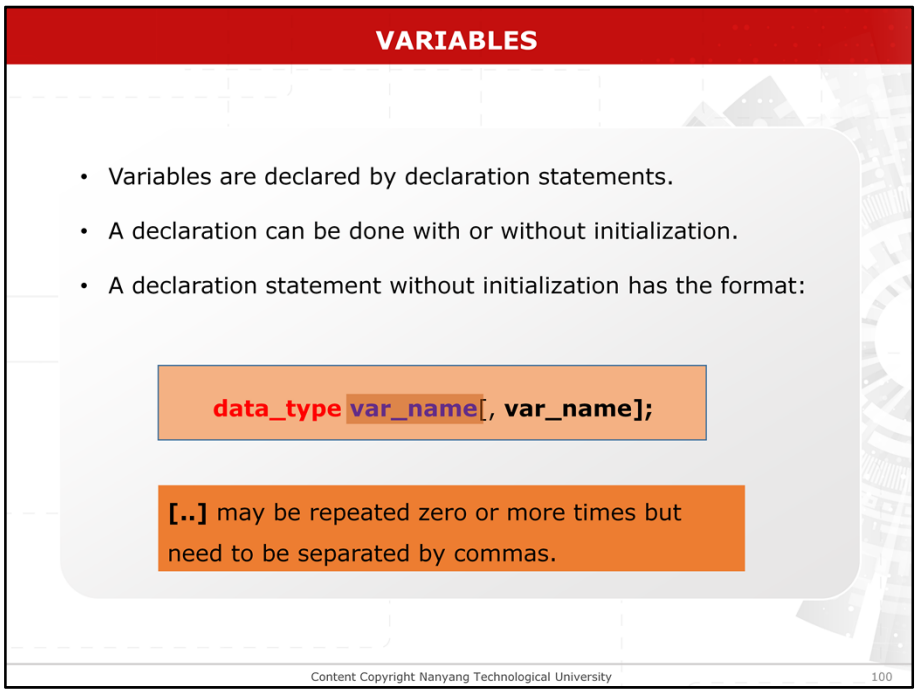
As shown



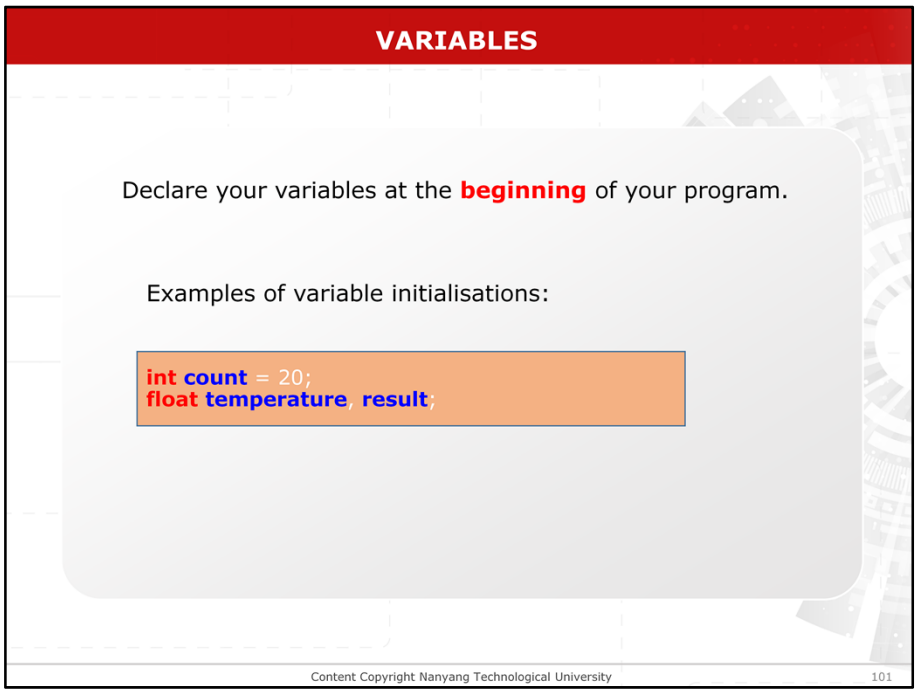
where data_type can be **int**, **car**, **float**, **double**, etc.



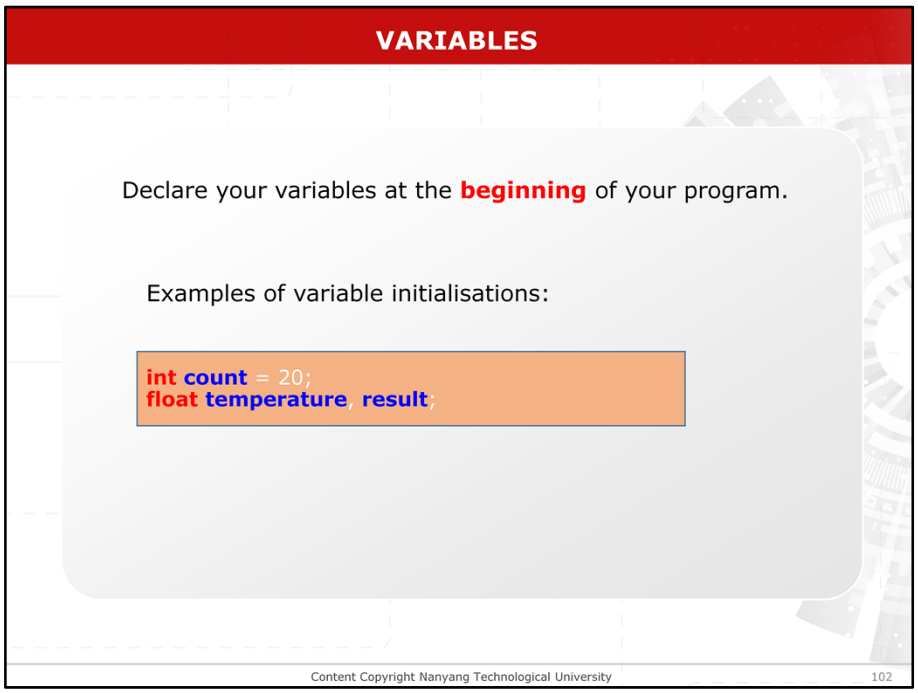
var_name is the name of the variable.



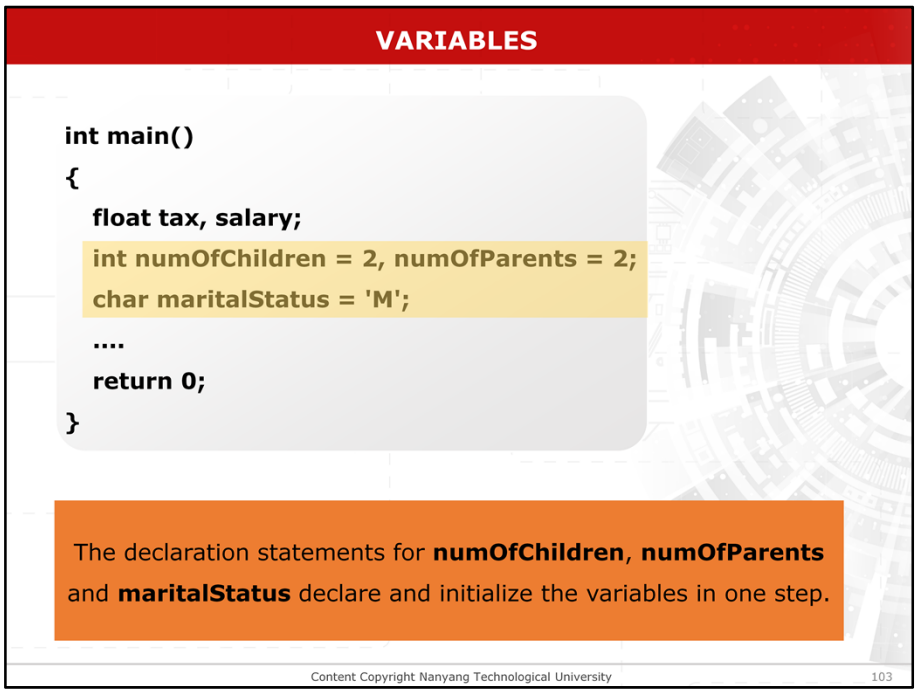
[..] may be repeated zero or more times but need to be separated by commas.



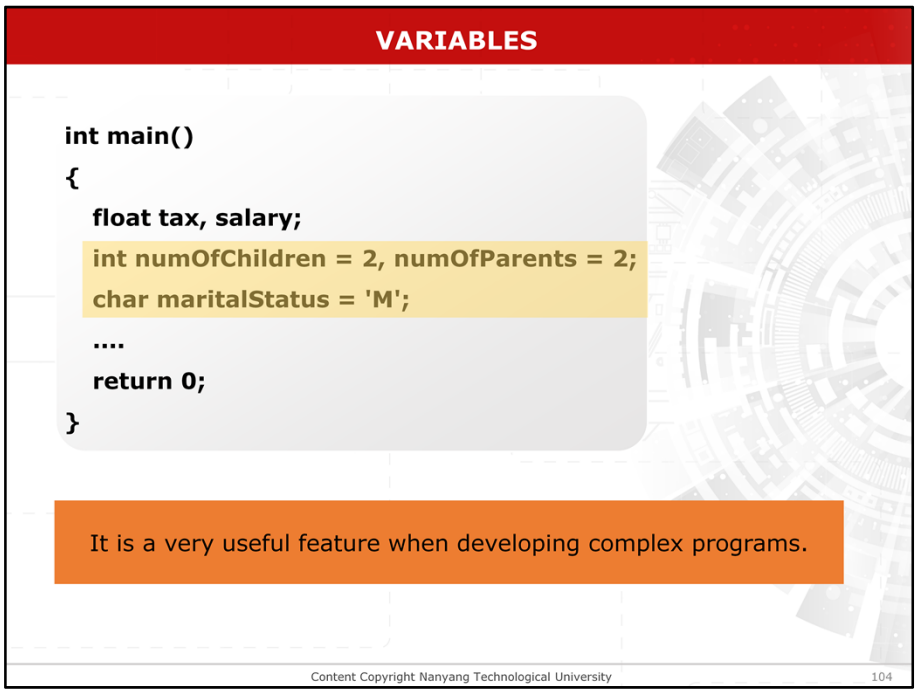
During program execution, memory storage of suitable size is assigned for each variable.
A variable must be declared first before it is used.



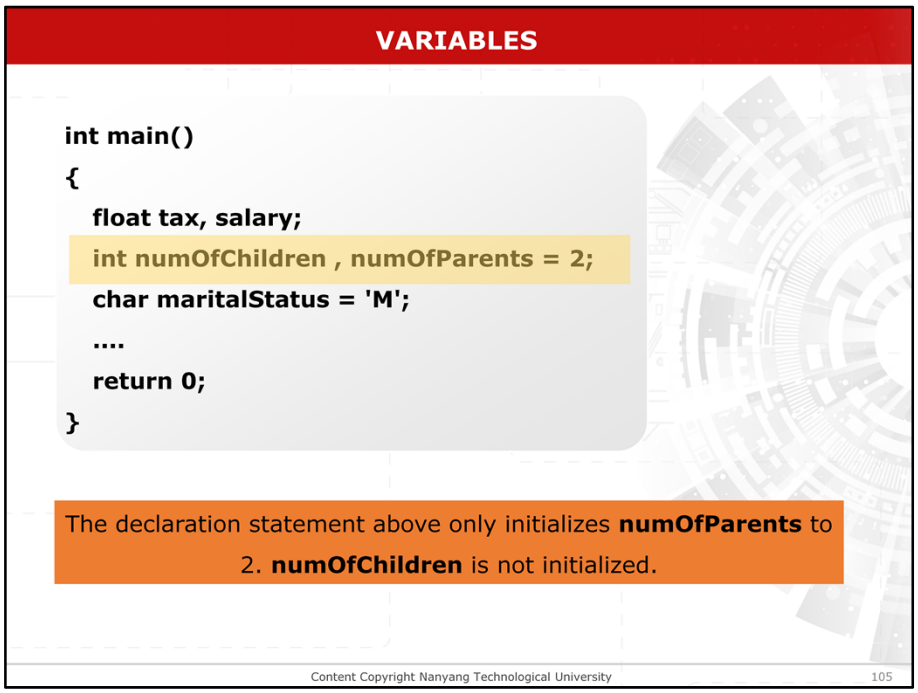
The memory location is used to store the current value of the variable. Variable names can then be used to retrieve the value stored in the memory location.
Initialization can also be done as part of a declaration. This means that a variable is given a starting value when it is declared.



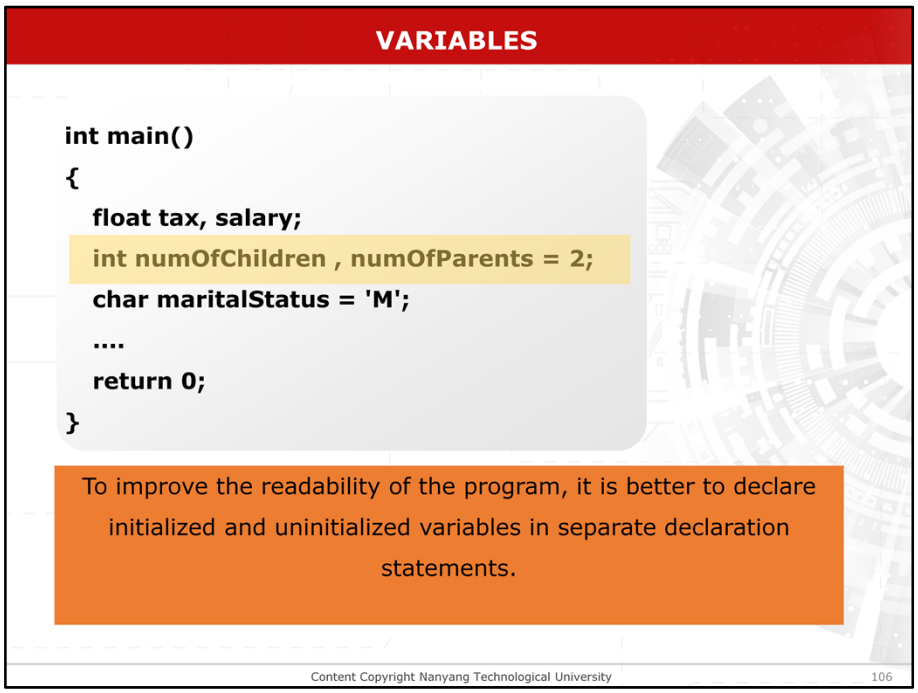
In the following program,
The declaration statements for **numOfChildren**, **numOfParents** and **maritalStatus** declare and initialize the variables in one step.



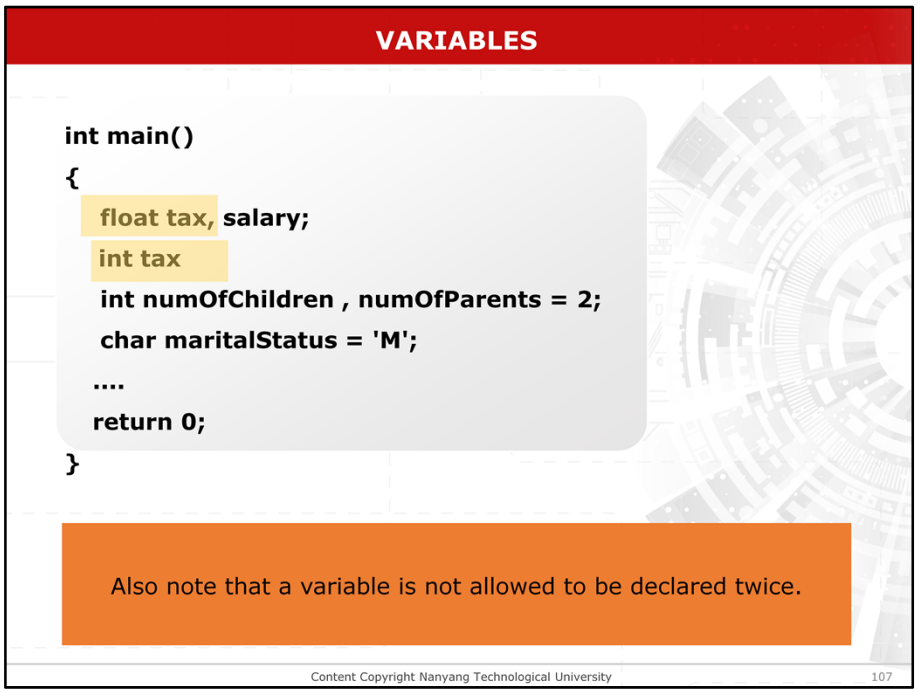
It is a very useful feature when developing complex programs.



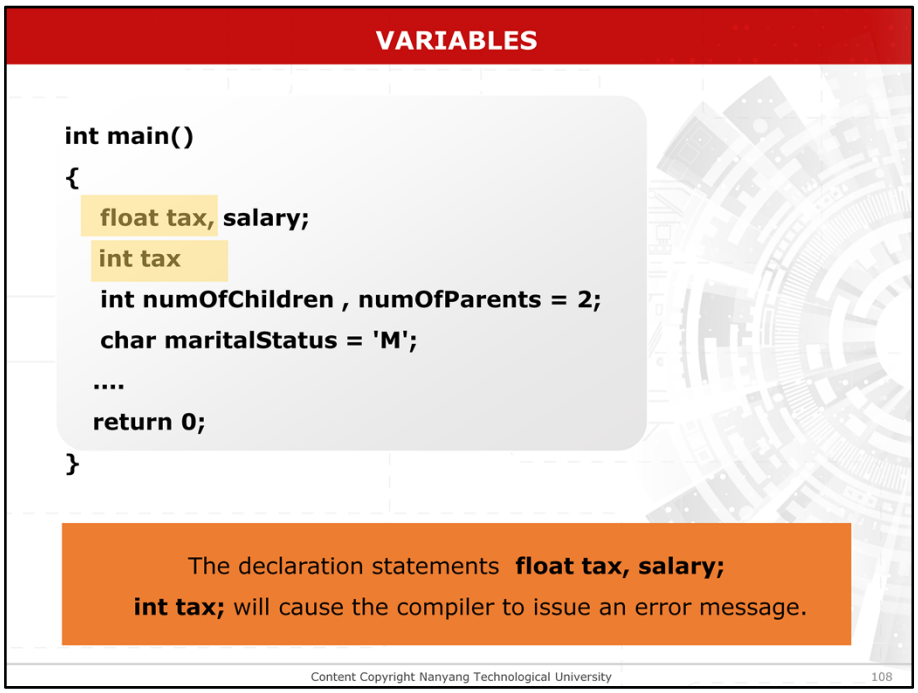
The declaration statement shown here only initializes **numOfParents** to 2. **numOfChildren** is not initialized.



To improve the readability of the program, it is better to declare initialized and uninitialized variables in separate declaration statements.



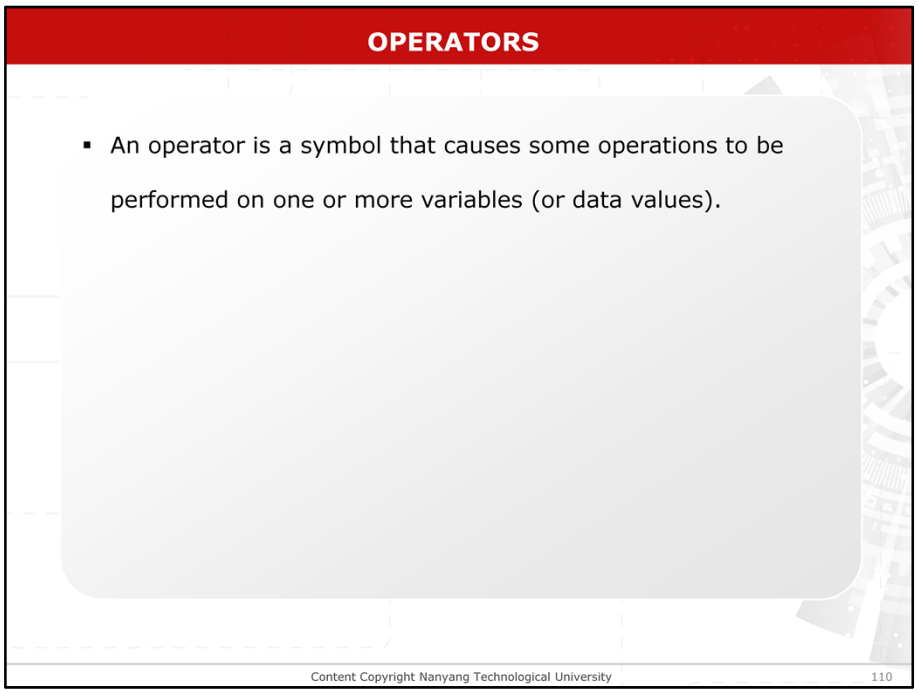
Also note that a variable is not allowed to be declared twice.



The declaration statements fl**oat tax, salary;** **int tax;** will cause the compiler to issue an error message.

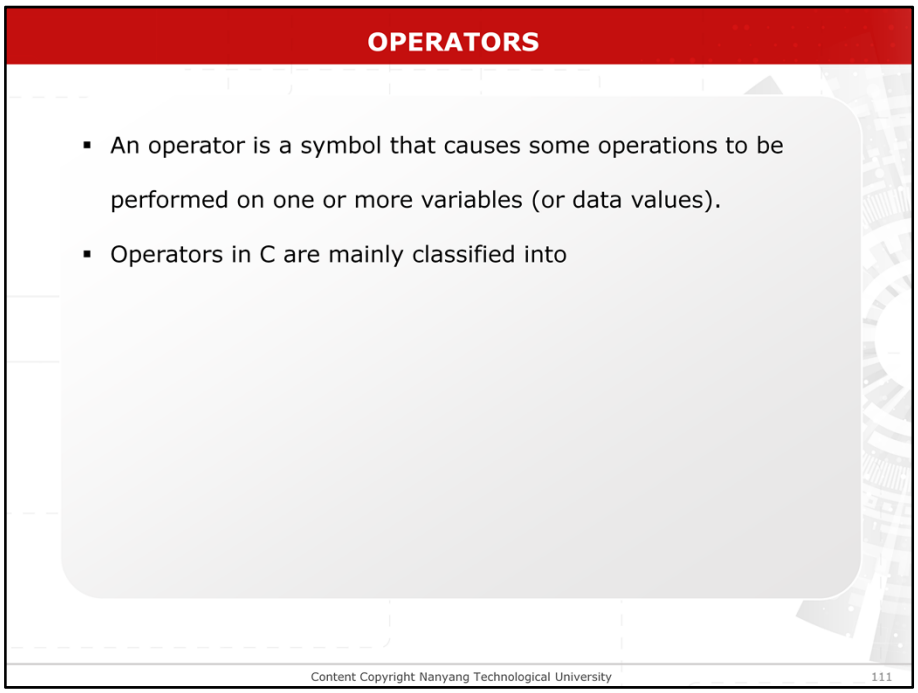

OPERATORS
Content Copyright Nanyang Technological University
109

Operators



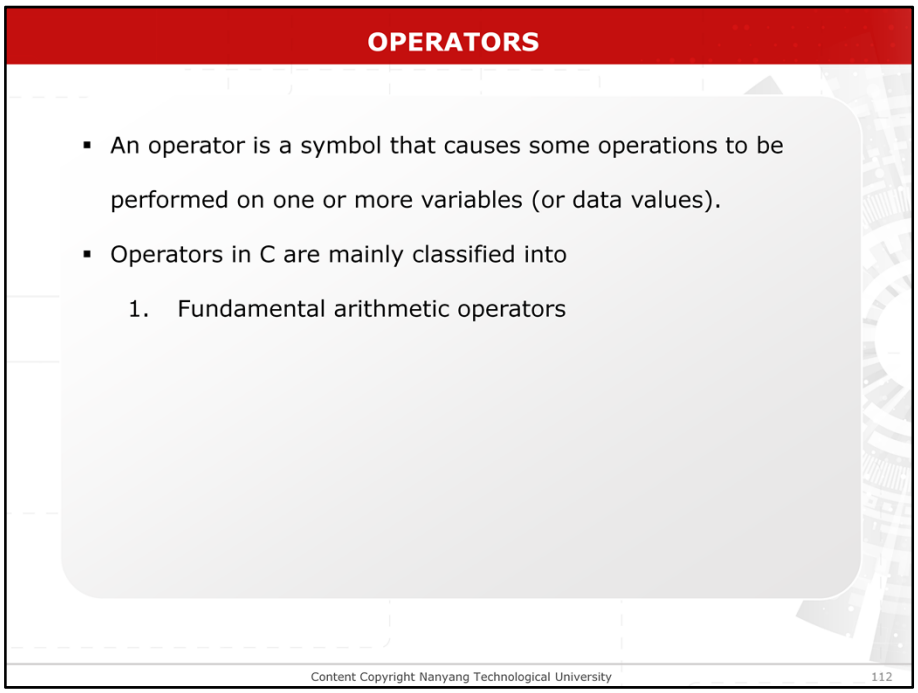
Operators

An operator is a symbol that causes some operations to be performed on one or more variables (or data values).

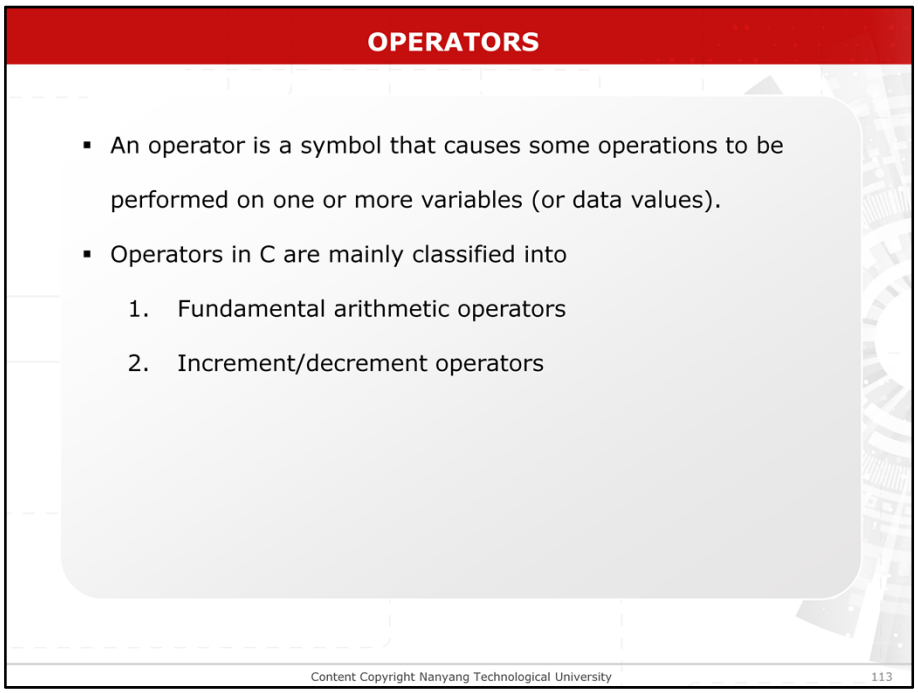


Operators

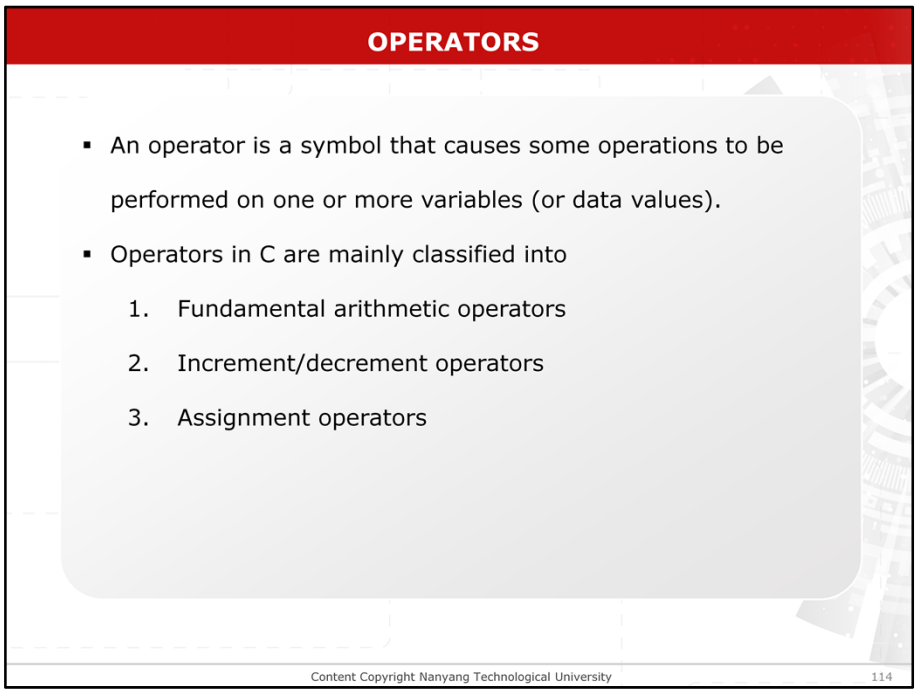
Operators in C are mainly classified into



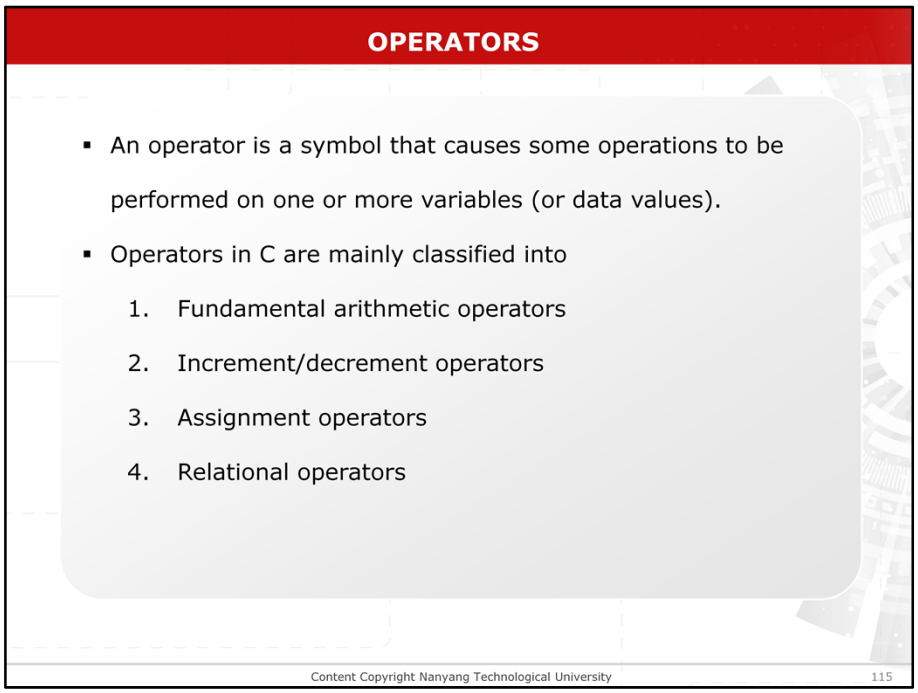
fundamental arithmetic operators



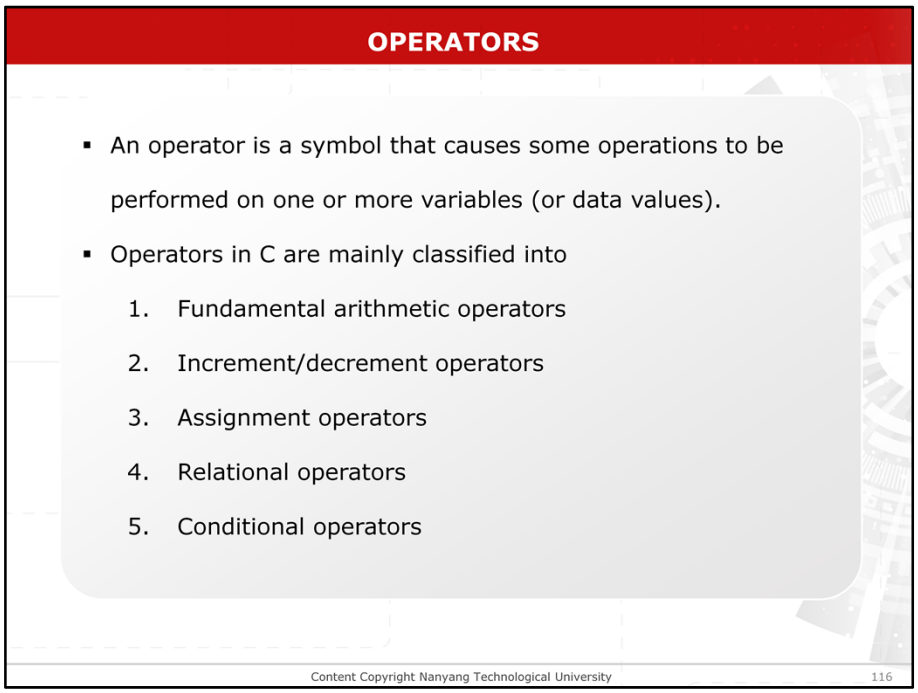
increment/decrement operators



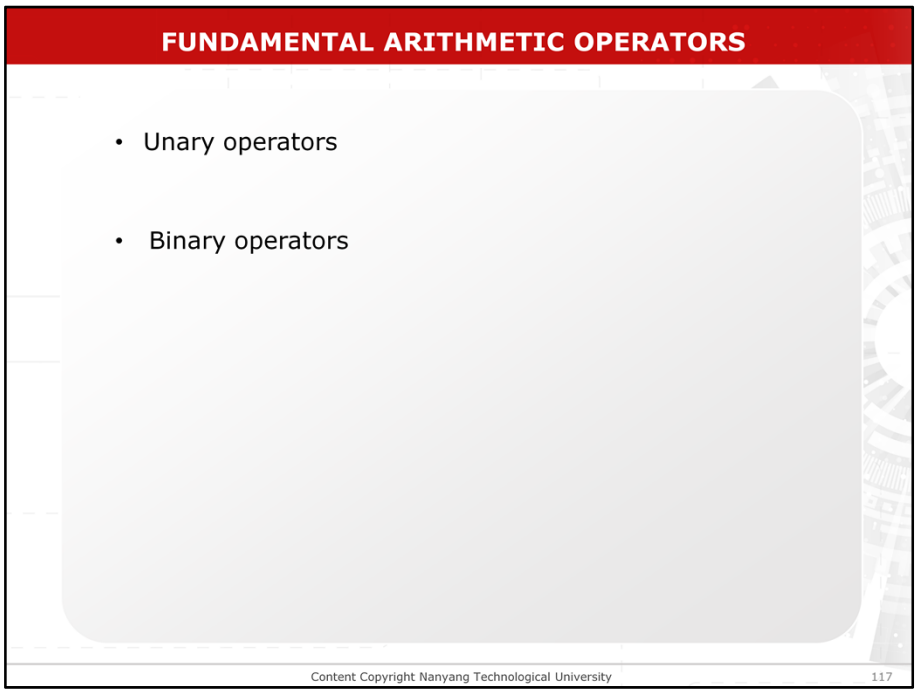
assignment operators



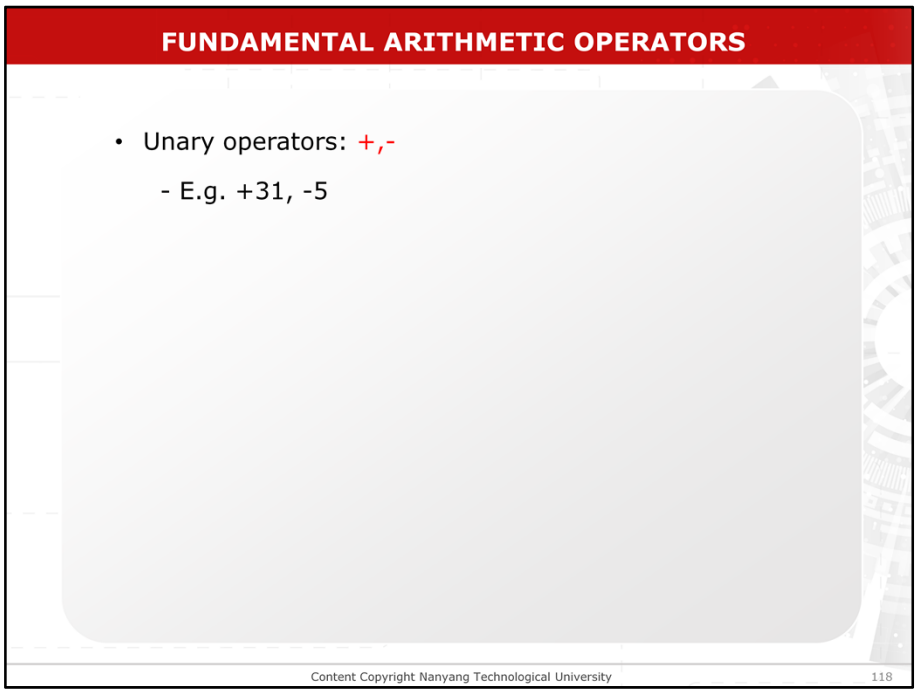
relational operatorS



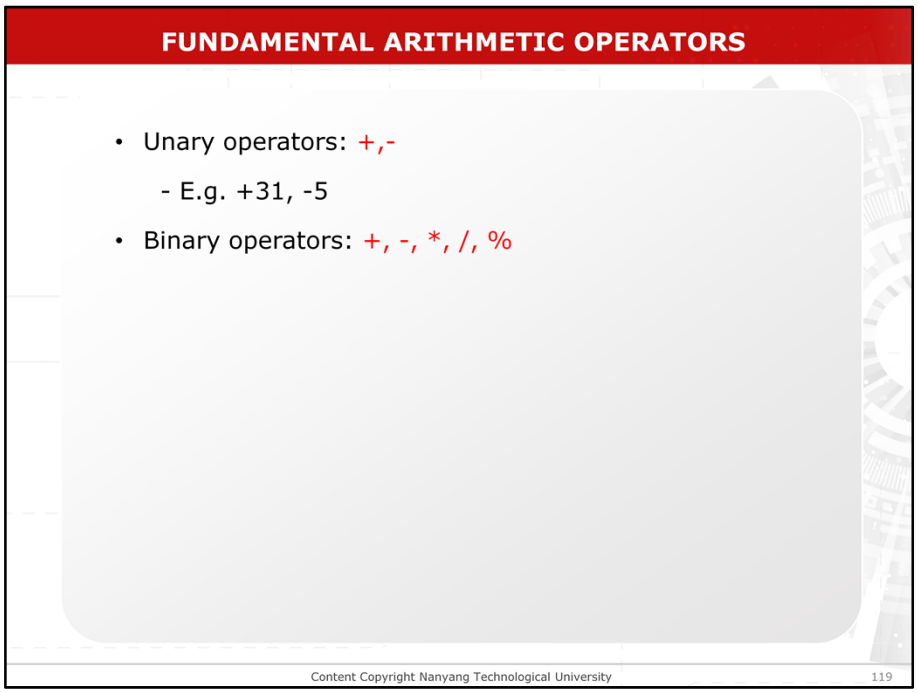
and conditional operators. Relational and conditional operators will be discussed in the next lecture.



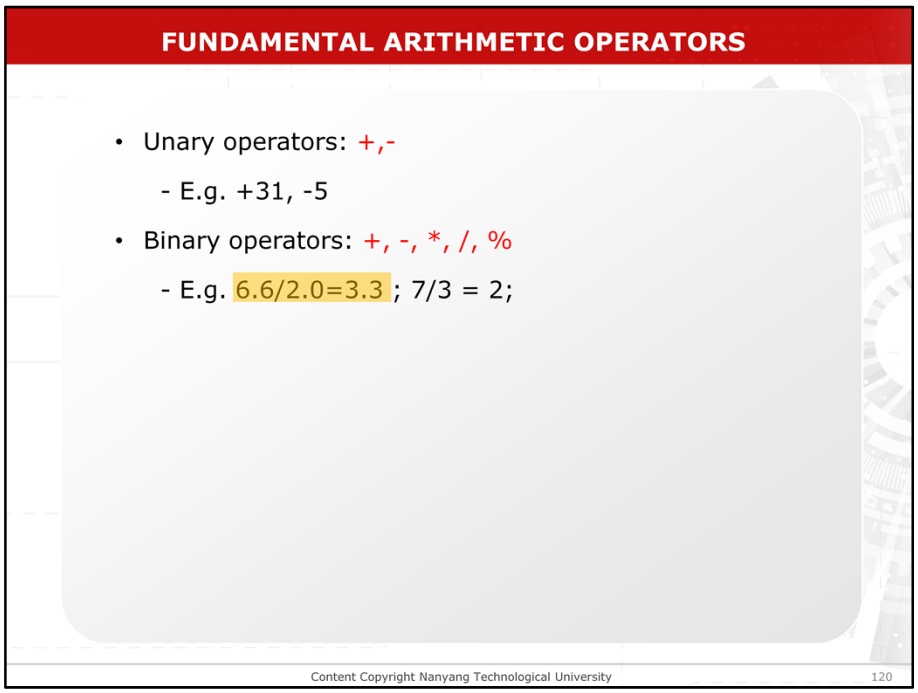
Fundamental Arithmetic Operators
unary operators and binary operators ARE TWO TYPES OF FUNDAMENTAL ARITHMETIC OPERATORS



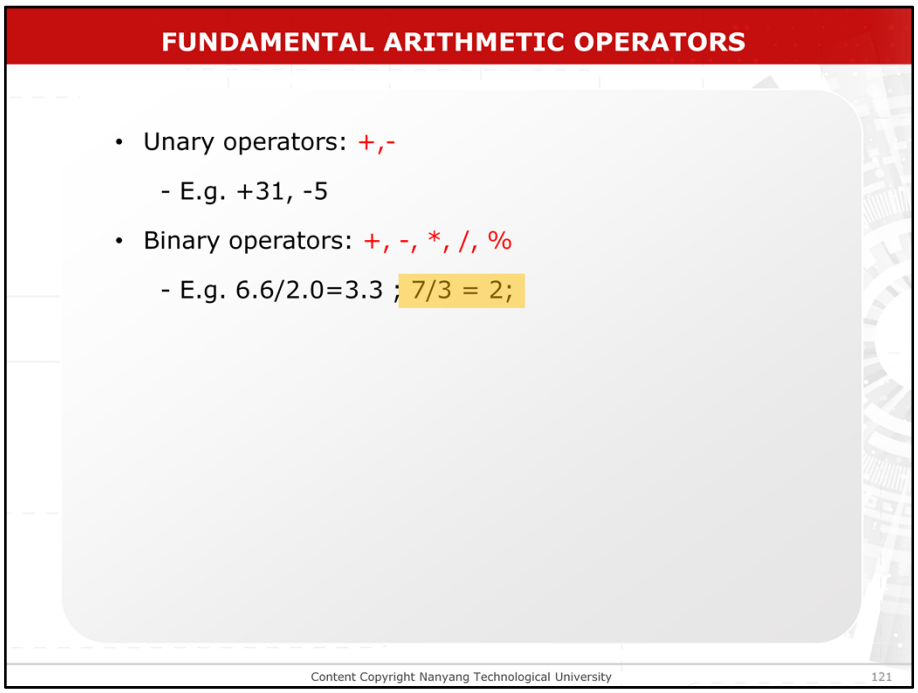
Unary operators are positive, negative e.g. POSITIVE 31, NEGATIVE 5. They can be used to change the sign of a value



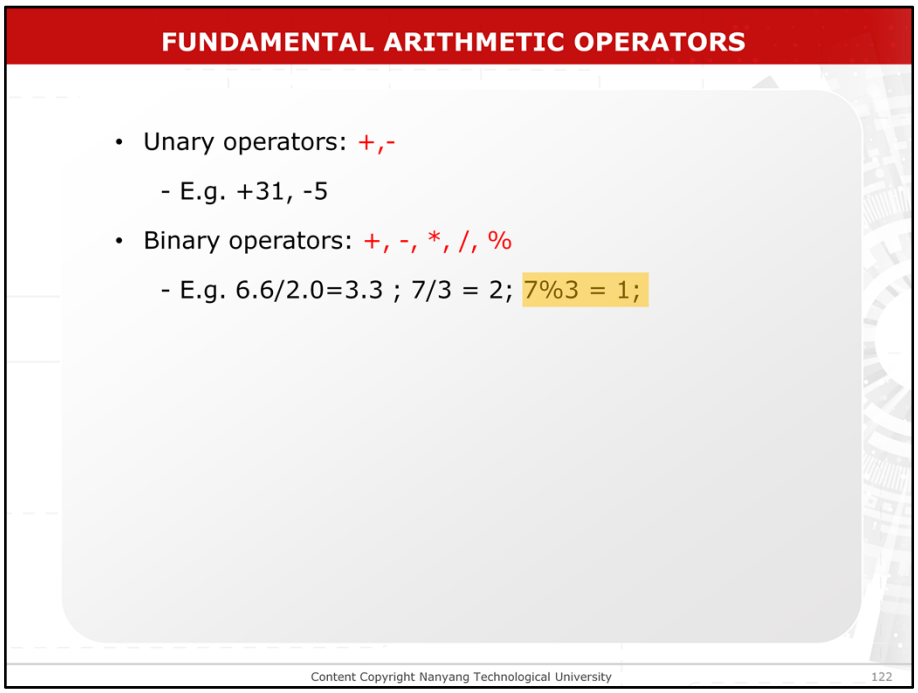
Binary operators are addition (**+**), subtraction (**-**), multiplication (*****), division (**/**) and modulus (**%**).



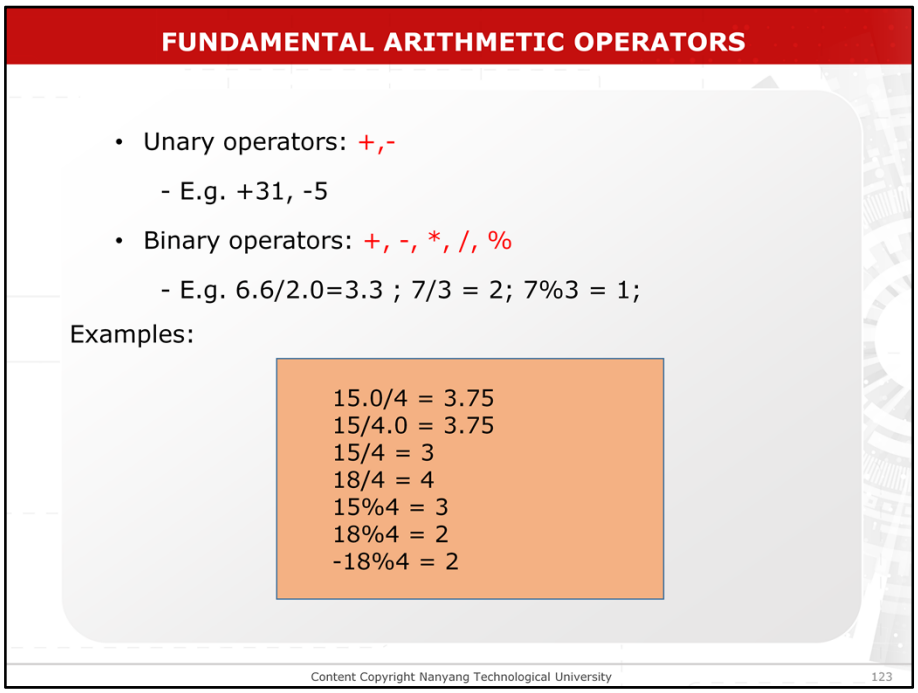
Fundamental Arithmetic Operators
Note that the division of floating point numbers (or one floating point number with an integer) returns a floating point number.



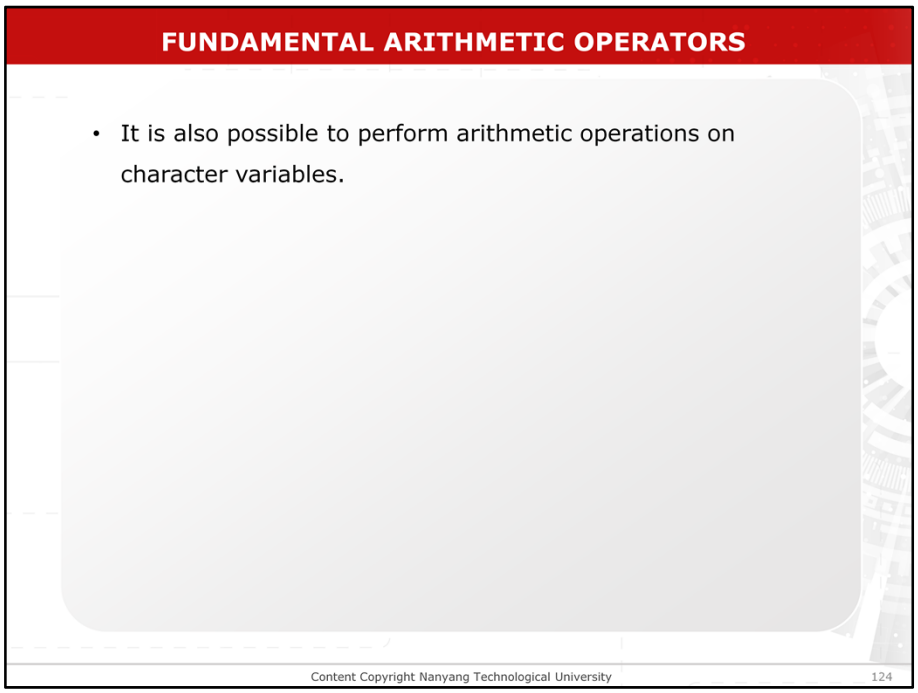
However, the division operator (**/**) for integers returns integer results.



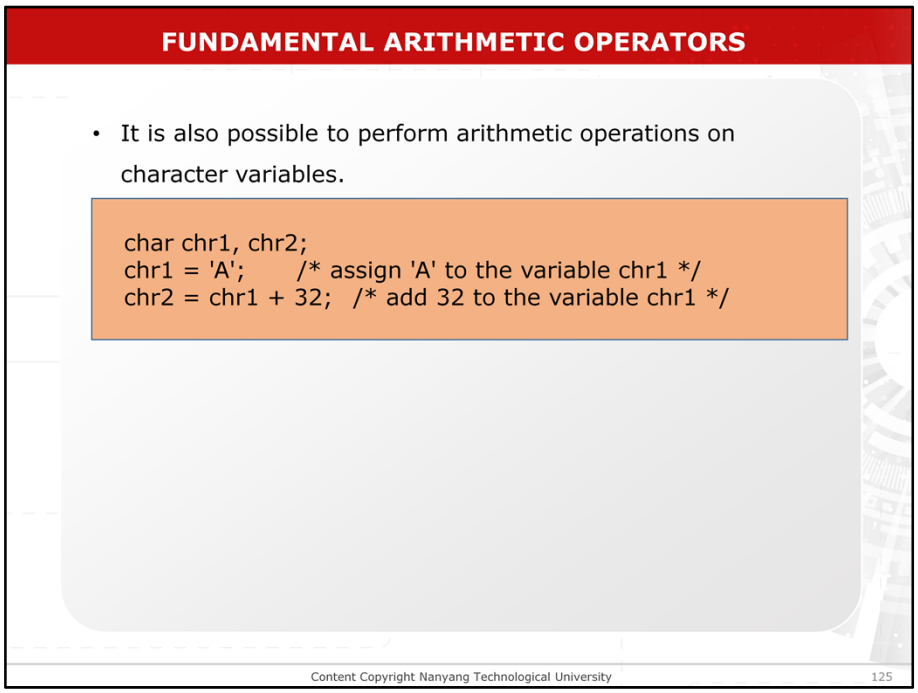
The modulus operator (**%**) returns the remainder of an integer division.



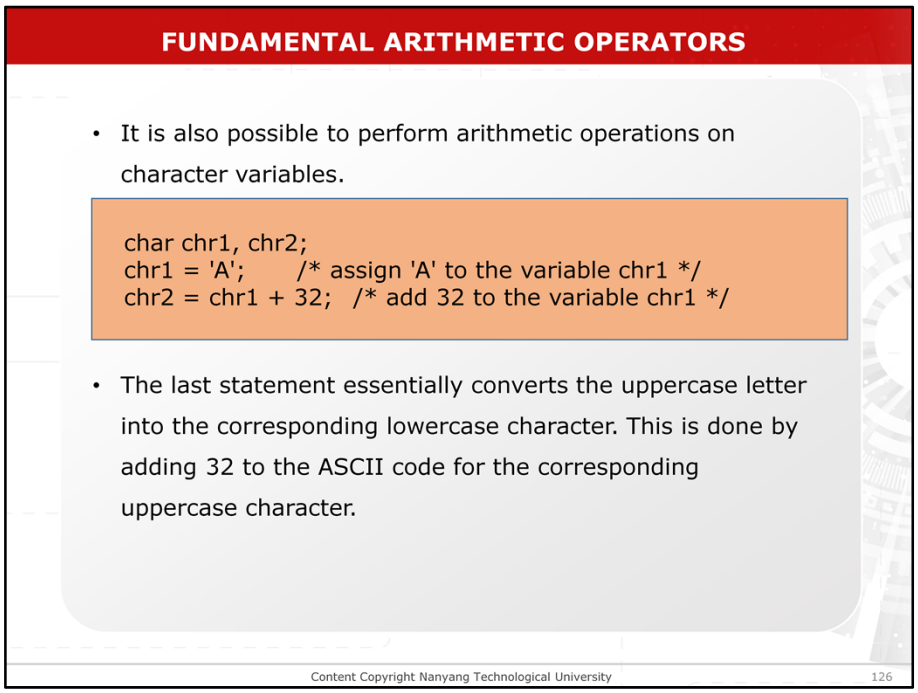
Few examples are shown here.



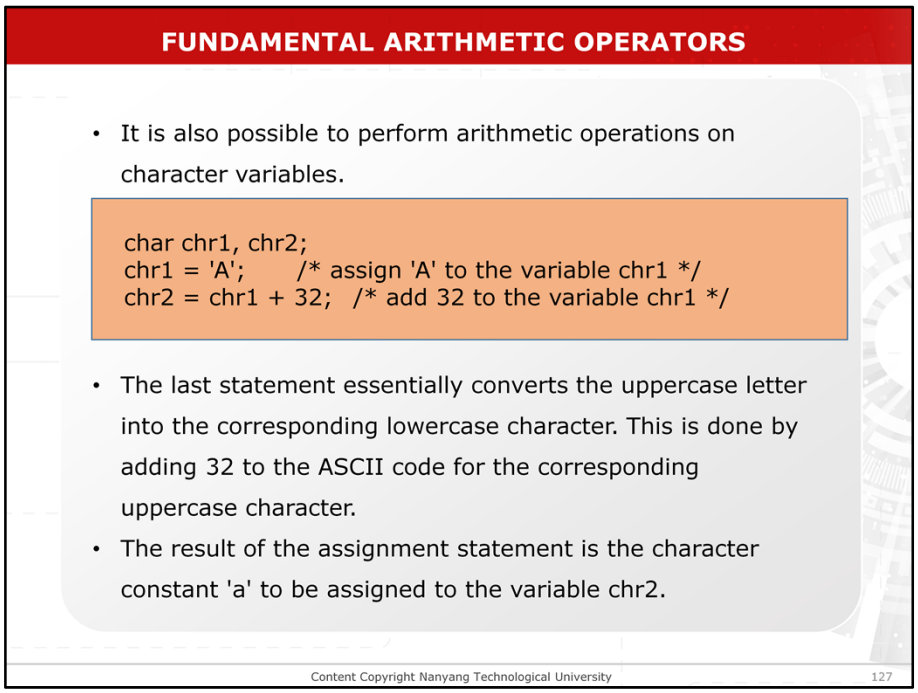
It is also possible to perform arithmetic operations on character variables.



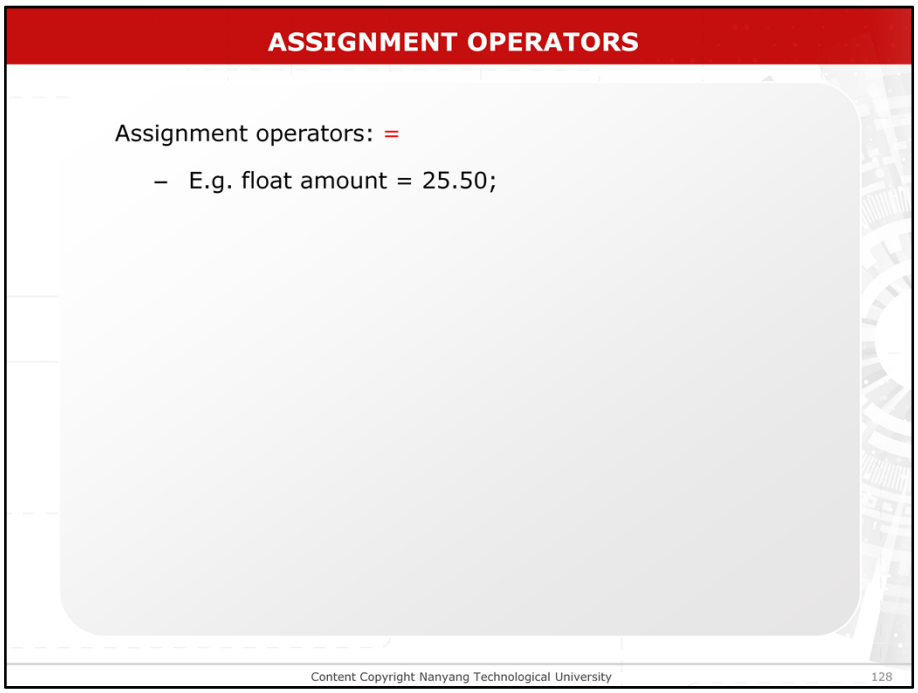
Few examples are shown here.



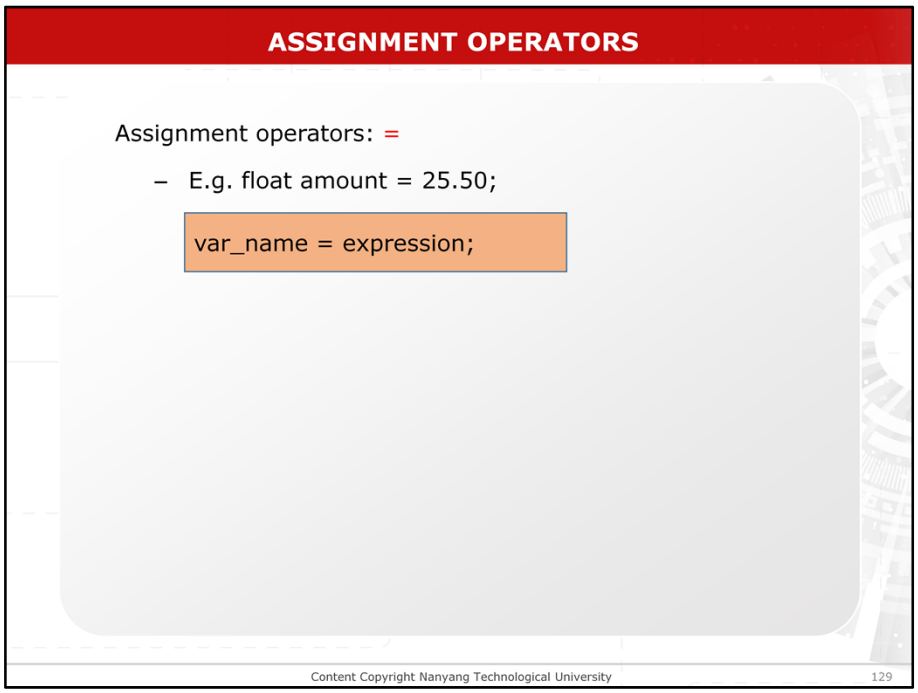
The last statement essentially converts the uppercase letter into the corresponding lowercase character. This is done by adding 32 to the ASCII code for the corresponding uppercase character.



The result of the assignment statement is the character constant '**eh**' to be assigned to the variable **chr2**. This is a very useful technique in converting an uppercase letter to its lowercase counterpart, or vice versa.



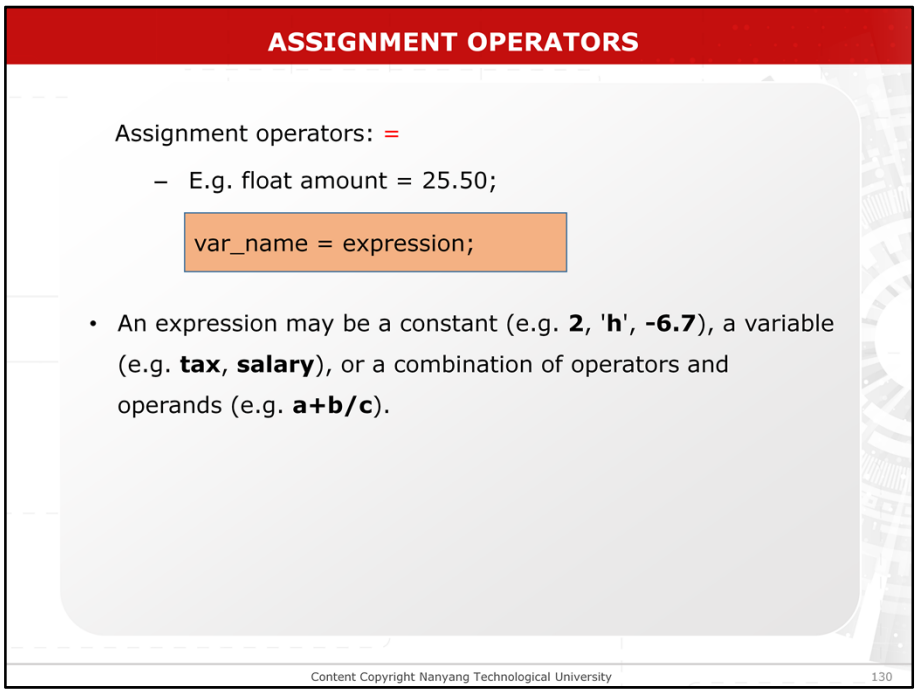
Assignment Operators
An assignment operator (**=**) is used to assign a value to a variable. It is the basic building block of a program.



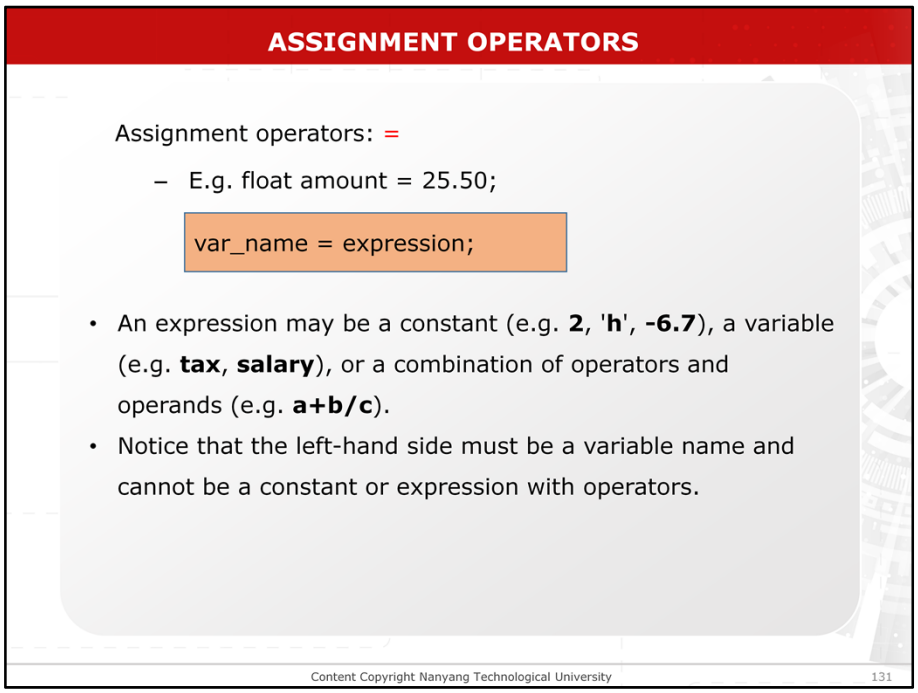
The value of a variable may be changed by the assignment statement in the following format:

var_name = expression;

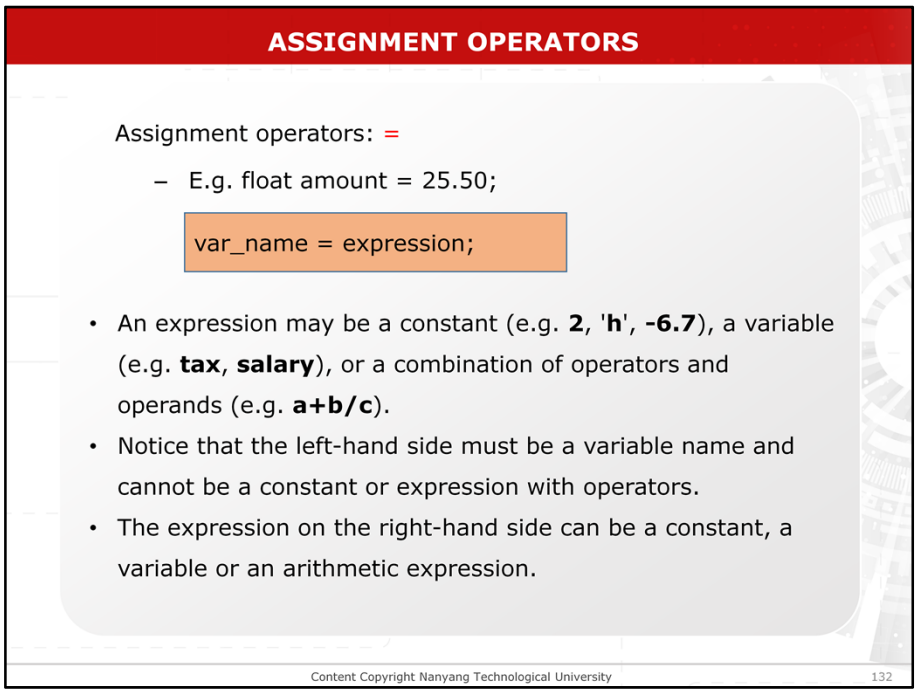
where **var_name** is a variable name, and the **expression** (on the right-hand side) provides a value that is assigned to the variable **var_name** on the left-hand side.



An expression may be a constant (e.g. **2**, '**h**', **-6.7**), a variable (e.g. **tax**, **salary**), or a combination of operators and operands (e.g. **a+b/c**).



Notice that the left-hand side must be a variable name and cannot be a constant or expression with operators.



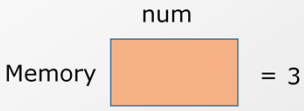
The expression on the right-hand side can be a constant, a variable or an arithmetic expression.

ASSIGNMENT OPERATORS

Consider the following assignment statement:

```
num = 3;
```

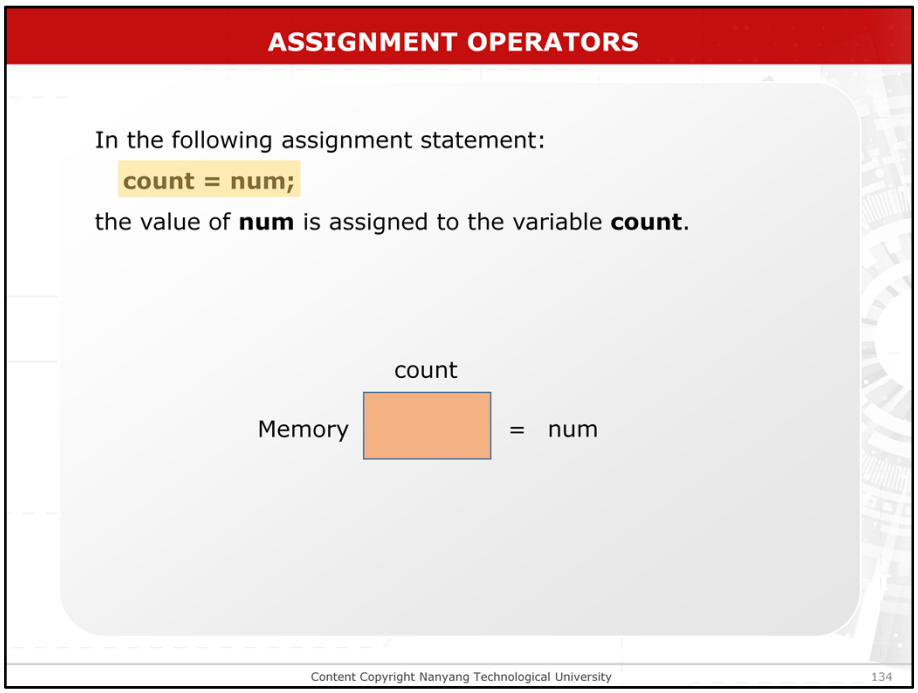
- The memory location of the variable **num** is used to store the value 3.
- The *value* of the variable refers to the value or content of the memory location, i.e. the actual value of the variable.



Consider the following assignment statement:

num = 3;

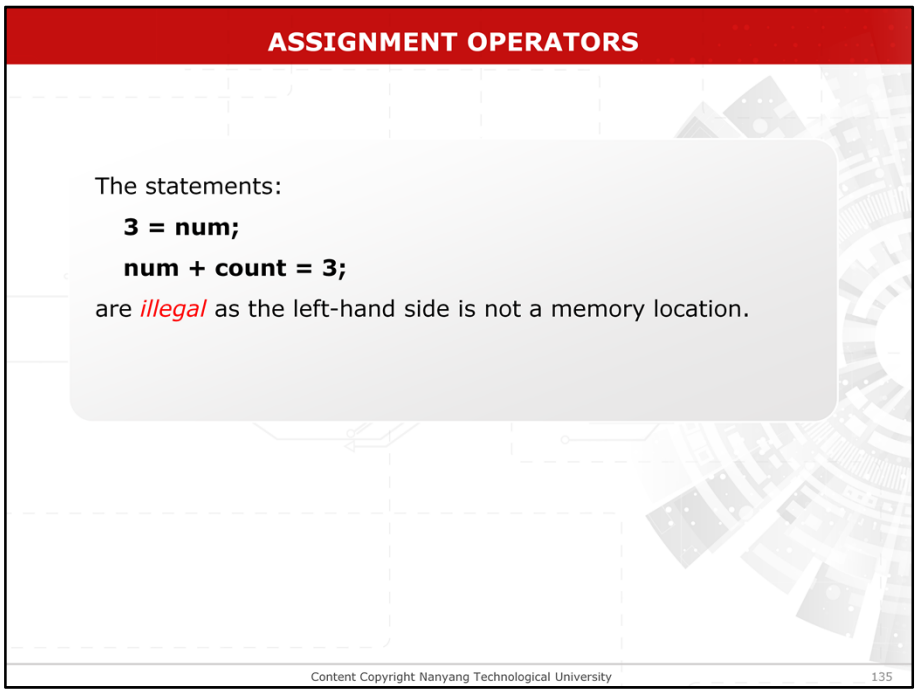
When a variable, e.g. **num**, is declared, the compiler allocates a memory location for storing the value of that variable. the memory location of the variable **num** is used to store the value 3. The *value* of the variable refers to the value or content of the memory location, i.e. the actual value of the variable.



In the following assignment statement:

count = num;

the value of **num** is assigned to the variable **count**.

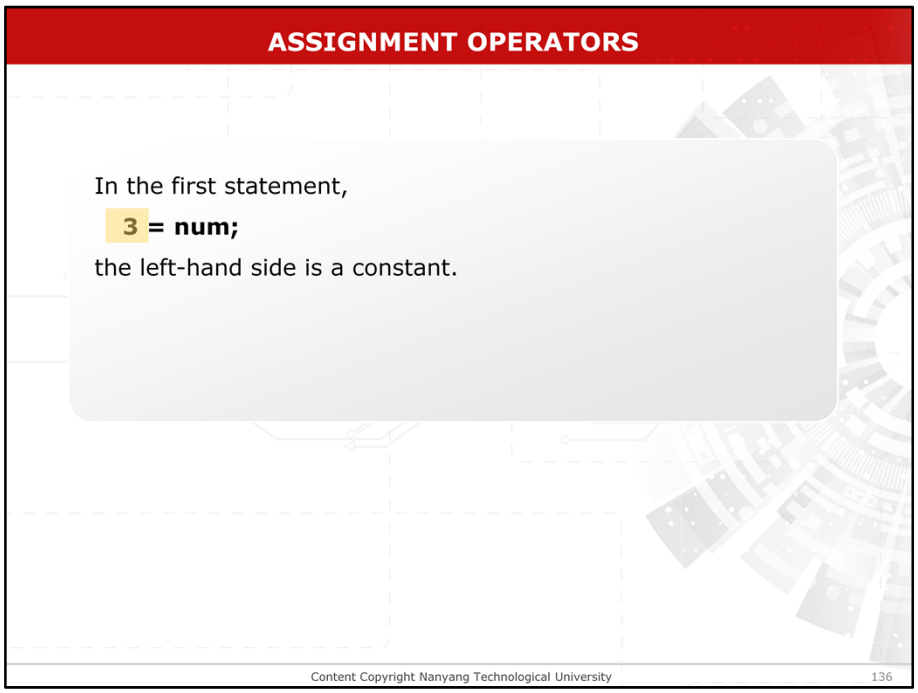


The statements

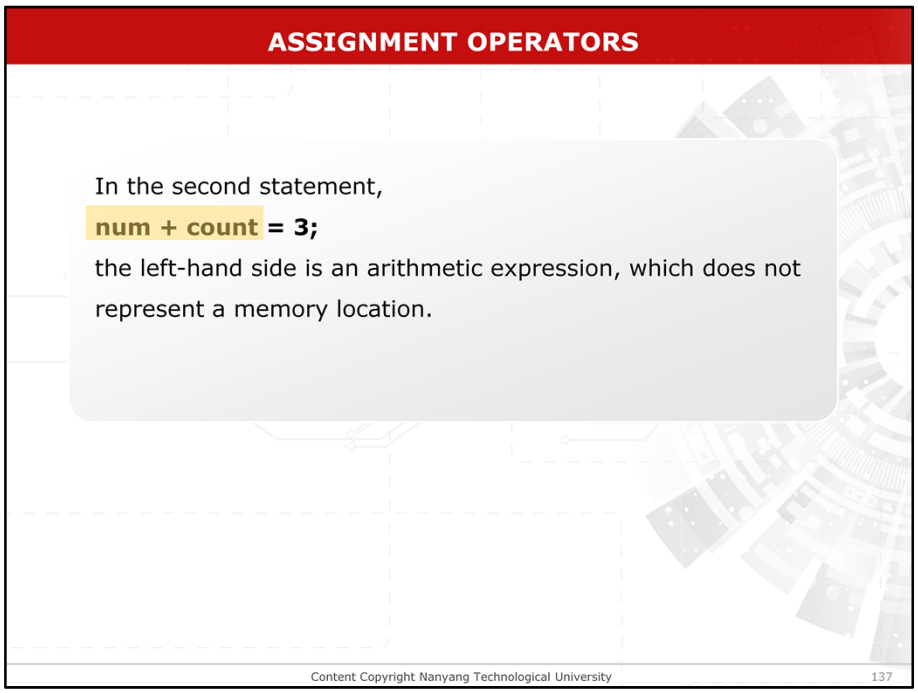
3 = num;

num + count = 3;

are *illegal* as the left-hand side is not a memory location



In the first statement, the left-hand side is a constant



and in the second statement, the left-hand side is an arithmetic expression, which does not represent a memory location.

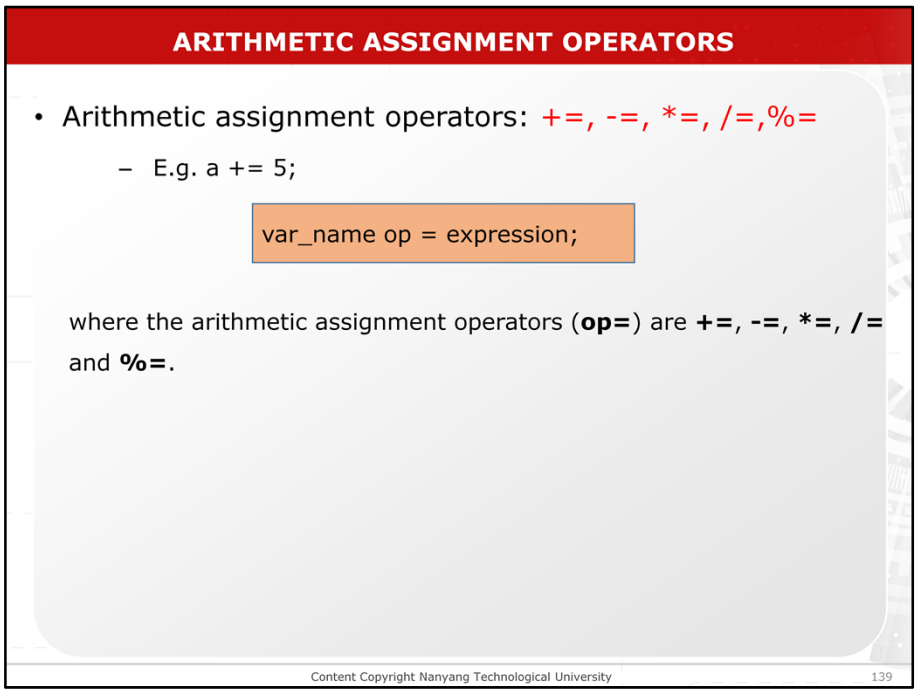
ARITHMETIC ASSIGNMENT OPERATORS

Arithmetic assignment operators: +=, -=, *=, /=,%=

- E.g. a += 5;

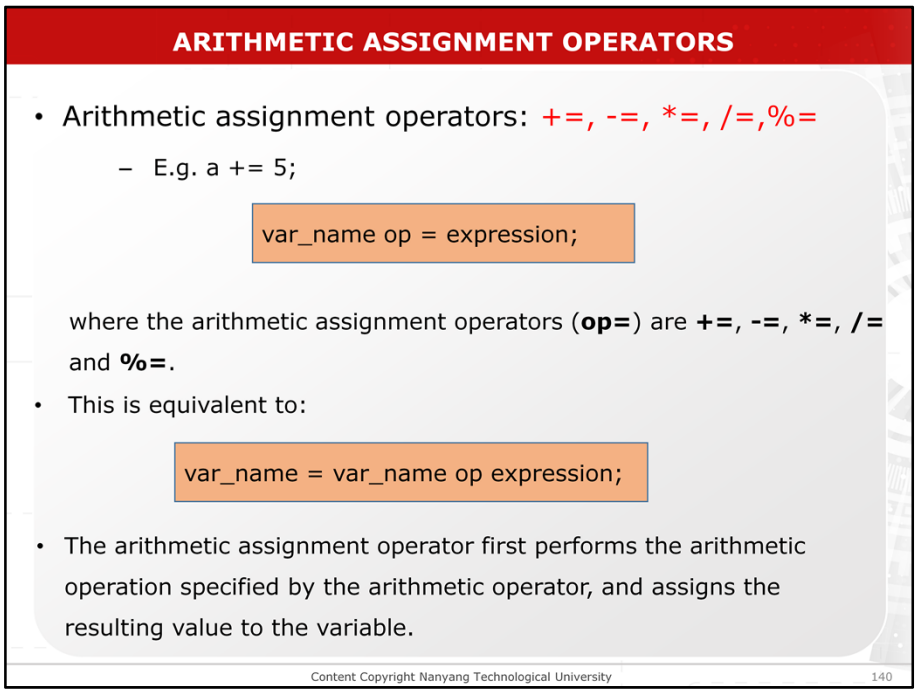
```
var_name op = expression;
```

where the arithmetic assignment operators (**op=**) are **+=**, **-=**, ***=**, **/=** and **%=**.

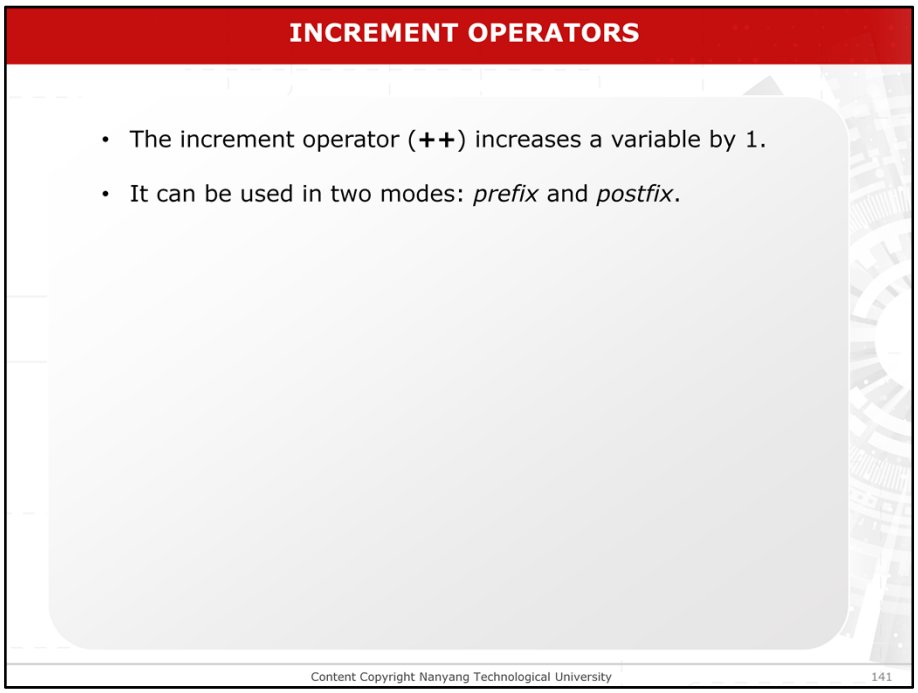


In addition to the assignment operators, C also has arithmetic assignment operators, which combine an arithmetic operator with the assignment operator. Arithmetic assignment operator statement has the format as shown:

where the arithmetic assignment operators are +=, minus equal, multiply equal, divide equal and modulus equal

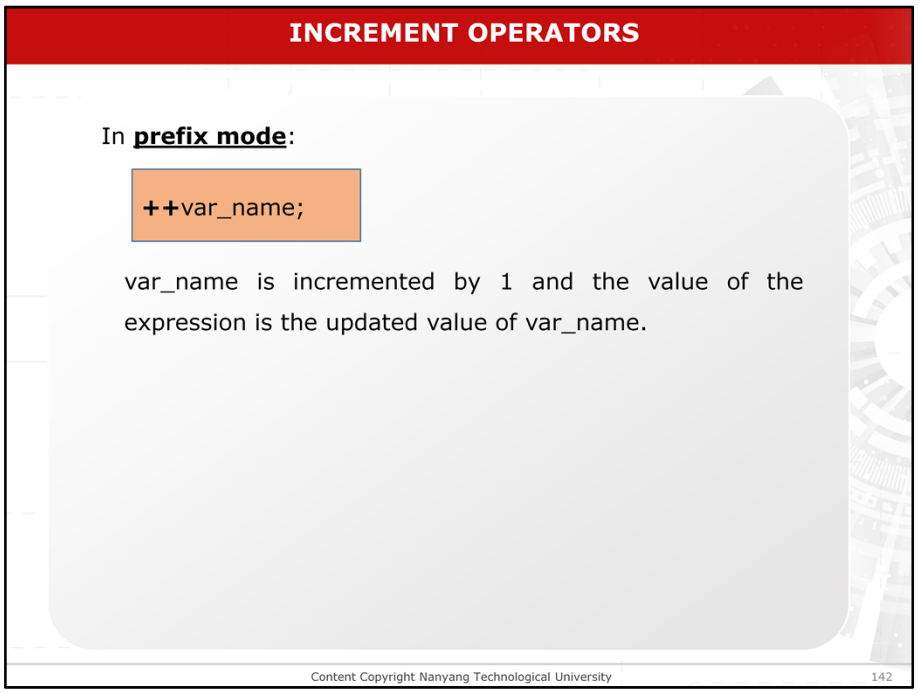


The arithmetic assignment operator first performs the arithmetic operation specified by the arithmetic operator, and assigns the resulting value to the variable.



Increment/decrement Operators

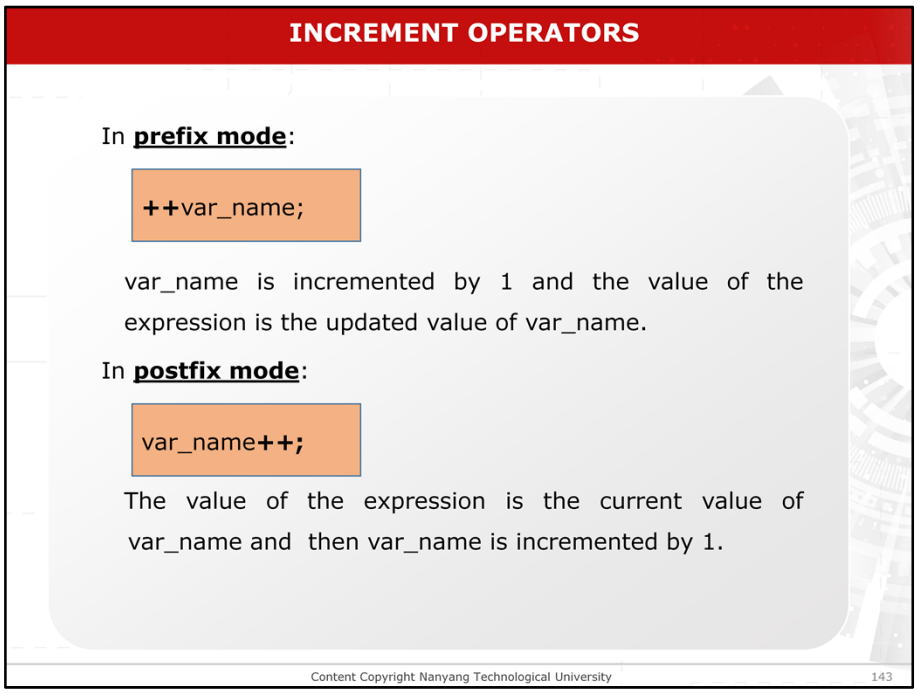
The increment operator (**++**) increases a variable by 1. It can be used in two modes: *prefix* and *postfix*.



The format of the prefix mode is

++var_name;

where **var_name** is incremented by 1 and the value of the expression is the updated value of **var_name**.



The format of the postfix mode is

var_name++;

where the value of the expression is the current value of **var_name** and then **var_name** is incremented by 1.

Notice that one important difference between the prefix and postfix modes is on the time when that operation is performed. In the prefix mode, the variable is incremented before any operation with it, while in the postfix mode, the variable is incremented after any operation with it.

INCREMENT OPERATORS

Code

```
#include <stdio.h>
int main()
{

  int num = 4;
  printf("value of num is %d\n", num);
  num++;  // ++num; i.e., num =num+1;
  printf("value of num is %d\n", num);
  num = 4;
  printf("value of num++ is %d\n", num++);
  printf("value of num is %d\n",num);
  printf("value of ++num is %d\n", ++num);
  printf("value of num is %d\n\n",num);
  return 0;

}
```

Output

In the example program, it shows some examples on the use of increment operators.

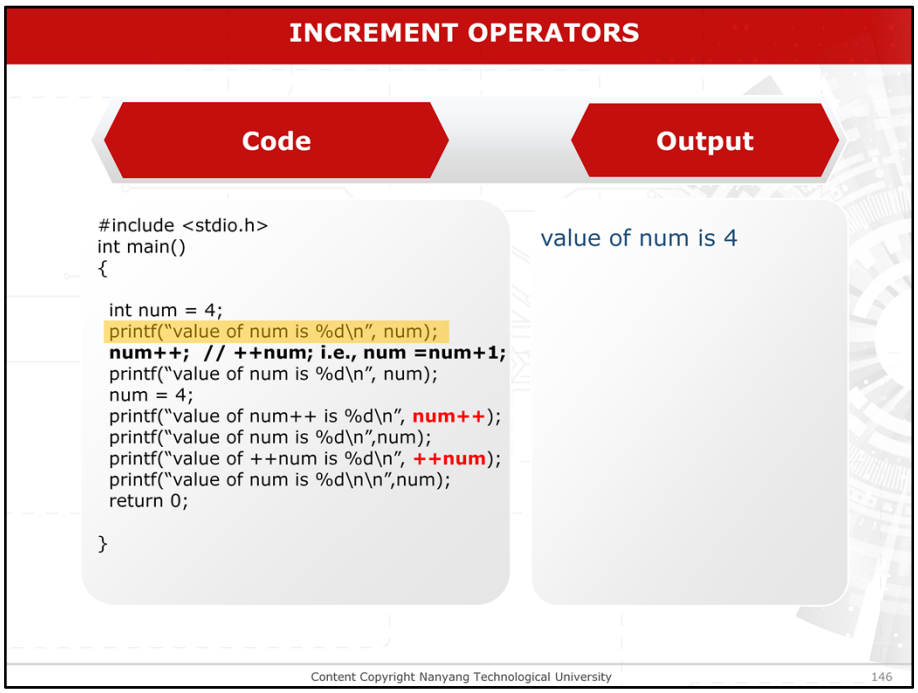
INCREMENT OPERATORS

Code

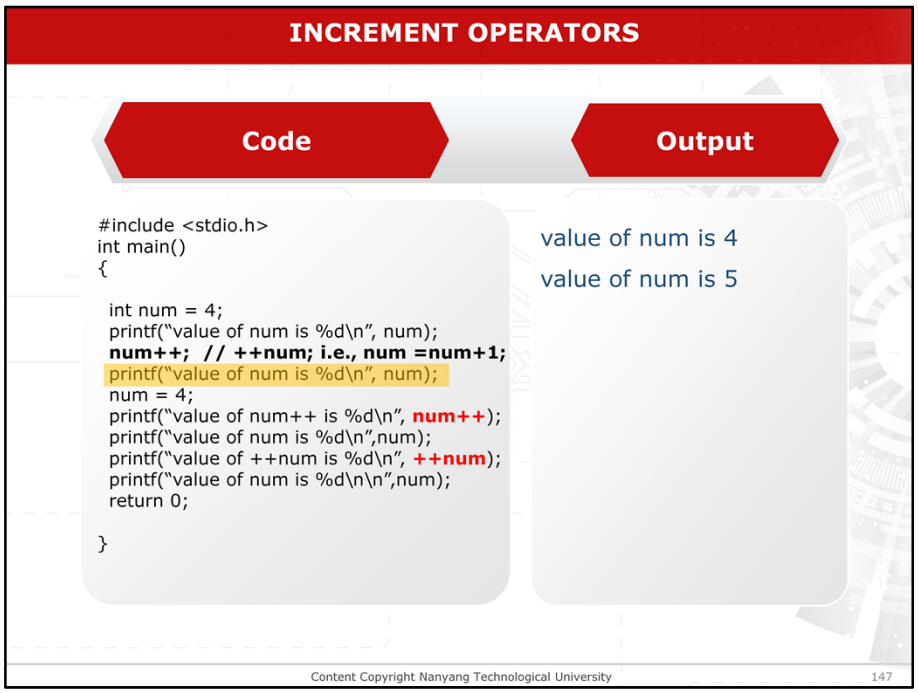
```
#include <stdio.h>
int main()
{

 int num = 4;
 printf("value of num is %d\n", num);
 num++;  // ++num; i.e., num =num+1;
 printf("value of num is %d\n", num);
 num = 4;
 printf("value of num++ is %d\n", num++);
 printf("value of num is %d\n",num);
 printf("value of ++num is %d\n", ++num);
 printf("value of num is %d\n\n",num);
 return 0;

}
```

Output

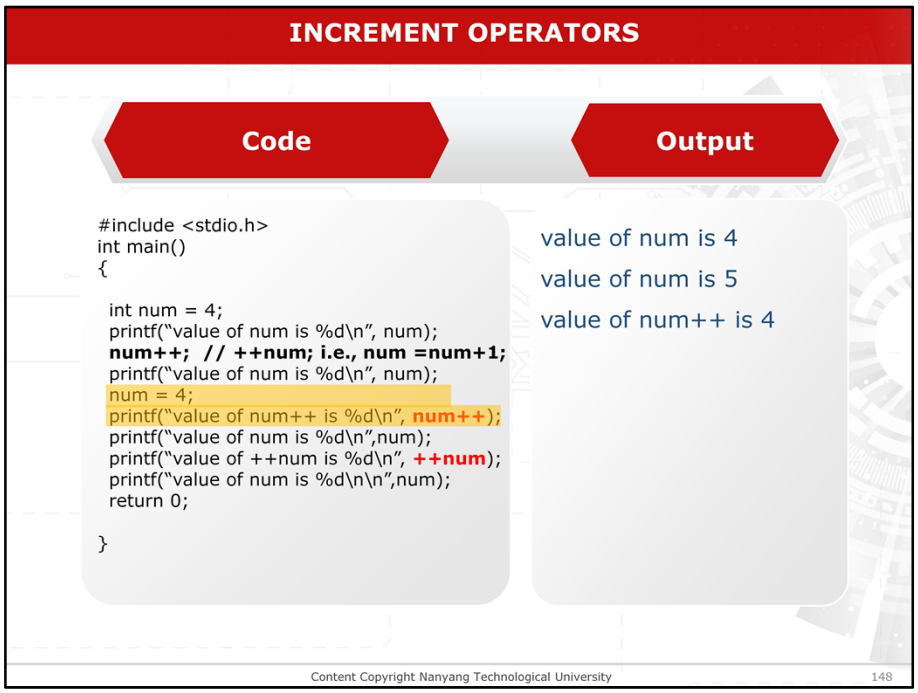
The initial value of the variable **num** is 4.



The first **printf()** statement prints the value of 4 for **num**.



The second **printf()** statement prints the value of 5 after increment. The variable **num** is assigned with the value 4 again.



As the third **printf()** statement uses the postfix mode, the value of **num** is incremented by 1 after the printing has taken place. Therefore, the third **printf()** statement prints the value of 4 for **num**.

INCREMENT OPERATORS

Code

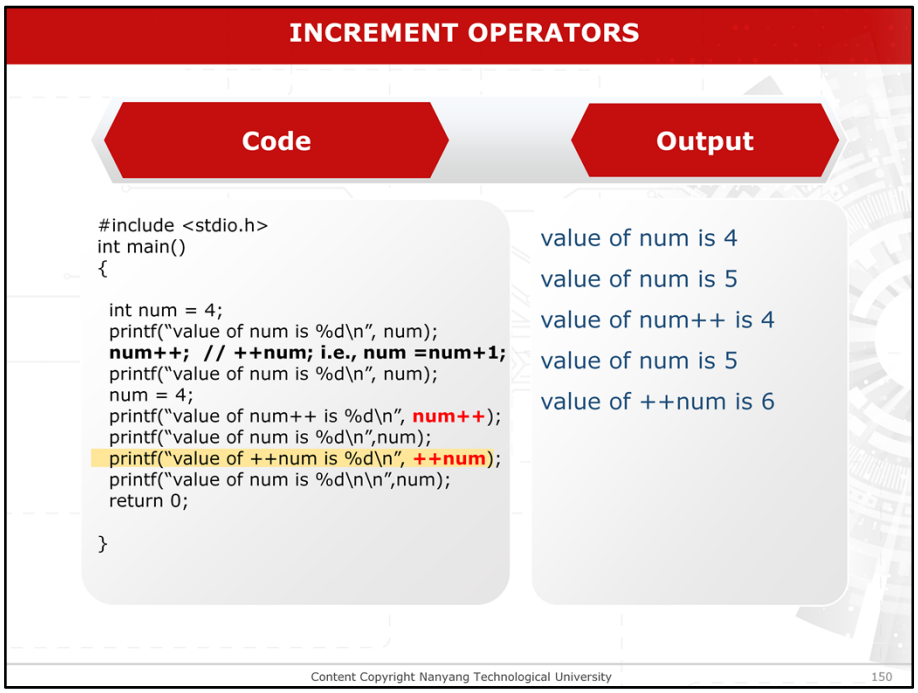
```
#include <stdio.h>
int main()
{

 int num = 4;
 printf("value of num is %d\n", num);
 num++;  // ++num; i.e., num =num+1;
 printf("value of num is %d\n", num);
 num = 4;
 printf("value of num++ is %d\n", num++);
 printf("value of num is %d\n",num);
 printf("value of ++num is %d\n", ++num);
 printf("value of num is %d\n\n",num);
 return 0;

}
```

Output

```
value of num is 4
value of num is 5
value of num++ is 4
value of num is 5
```

 149

The fourth statement then prints the value of 5.



The fifth **printf()** statement uses the prefix mode. The value of **num** is incremented before the printing is taken place. Thus, it prints the value 6.

INCREMENT OPERATORS

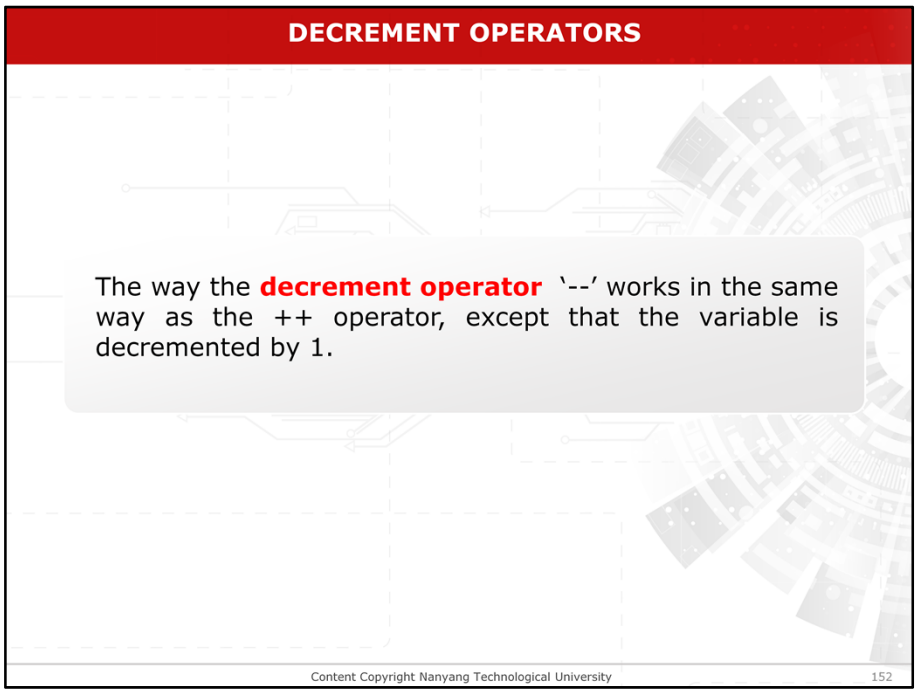
Code

```
#include <stdio.h>
int main()
{

 int num = 4;
 printf("value of num is %d\n", num);
 num++;  // ++num; i.e., num =num+1;
 printf("value of num is %d\n", num);
 num = 4;
 printf("value of num++ is %d\n", num++);
 printf("value of num is %d\n",num);
 printf("value of ++num is %d\n", ++num);
 printf("value of num is %d\n\n",num);
 return 0;

}
```

Output

```
value of num is 4
value of num is 5
value of num++ is 4
value of num is 5
value of ++num is 6
value of num is 6
```


The sixth statement also prints the value 6 for **num**.



The decrement operator (**--**) works in a similar way as the **++** operator, except that the variable is decremented by 1.

DECREMENT OPERATORS

Code

```
#include <stdio.h>
int main()
{

  int num = 4;
  printf("value of num is %d\n", num);
  num--;  // same as --num;
  printf("value of num is %d\n", num);
  num = 4;
  printf("value of num-- is %d\n", num--);
  printf("value of num is %d\n",num);
  printf("value of --num is %d\n", --num);
  printf("value of num is %d\n\n",num);
  return 0;

}
```

Output

In the example program, it shows some examples on the use of decrement operators.

DECREMENT OPERATORS

Code

```
#include <stdio.h>
int main()
{

 int num = 4;
 printf("value of num is %d\n", num);
 num--;  // same as --num;
 printf("value of num is %d\n", num);
 num = 4;
 printf("value of num-- is %d\n", num--);
 printf("value of num is %d\n",num);
 printf("value of --num is %d\n", --num);
 printf("value of num is %d\n\n",num);
 return 0;

}
```

Output

The initial value of the variable **num** is 4.

DECREMENT OPERATORS

Code

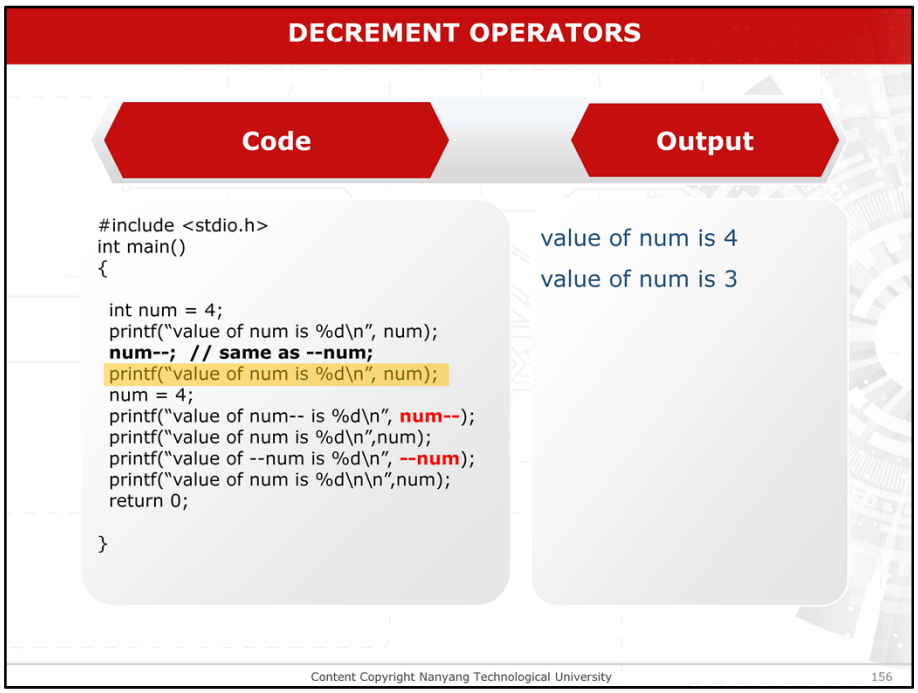
```
#include <stdio.h>
int main()
{

 int num = 4;
 printf("value of num is %d\n", num);
 num--;  // same as --num;
 printf("value of num is %d\n", num);
 num = 4;
 printf("value of num-- is %d\n", num--);
 printf("value of num is %d\n",num);
 printf("value of --num is %d\n", --num);
 printf("value of num is %d\n\n",num);
 return 0;

}
```

Output

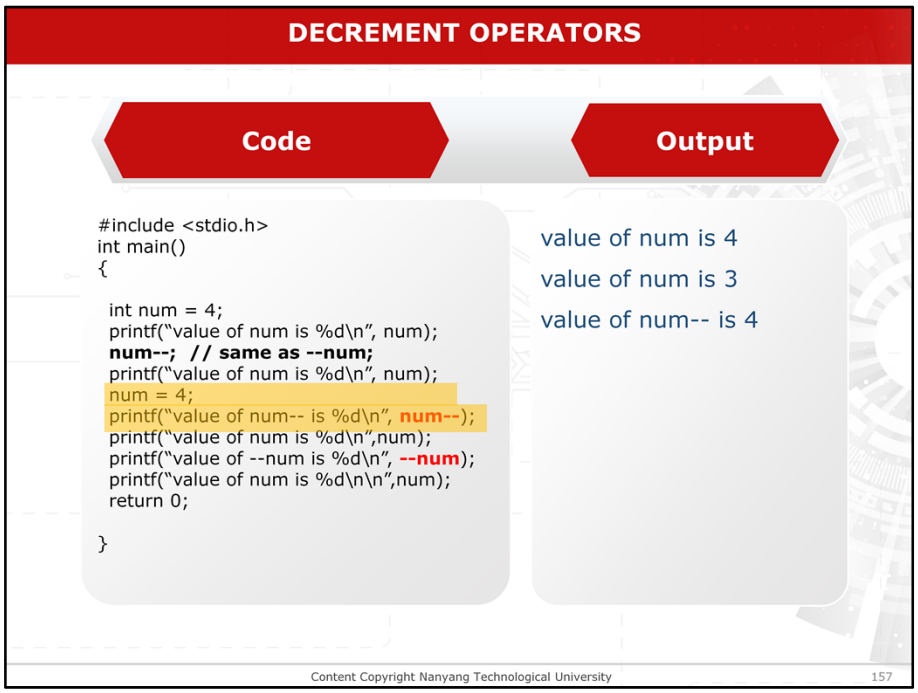
```
value of num is 4
```

 155

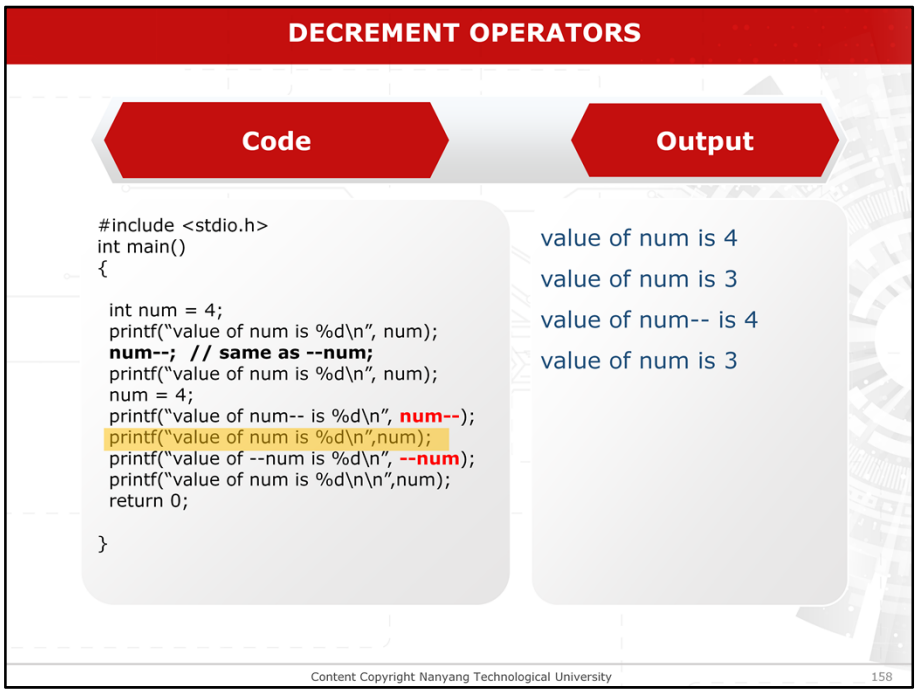
. The first **printf()** statement prints the value of 4 for **num**.



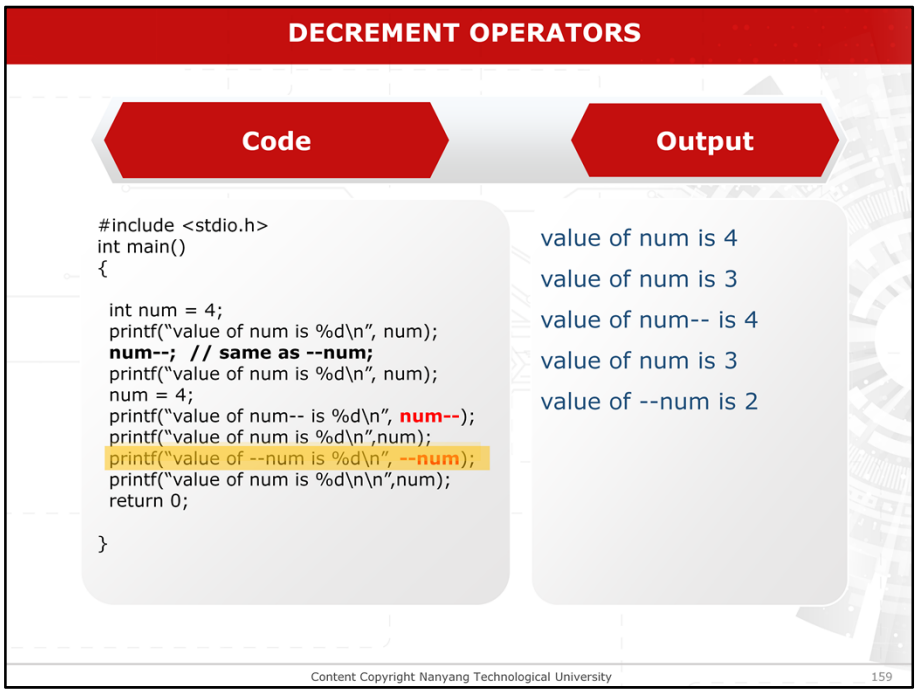
The second **printf()** statement prints the value of 3 after decrement.



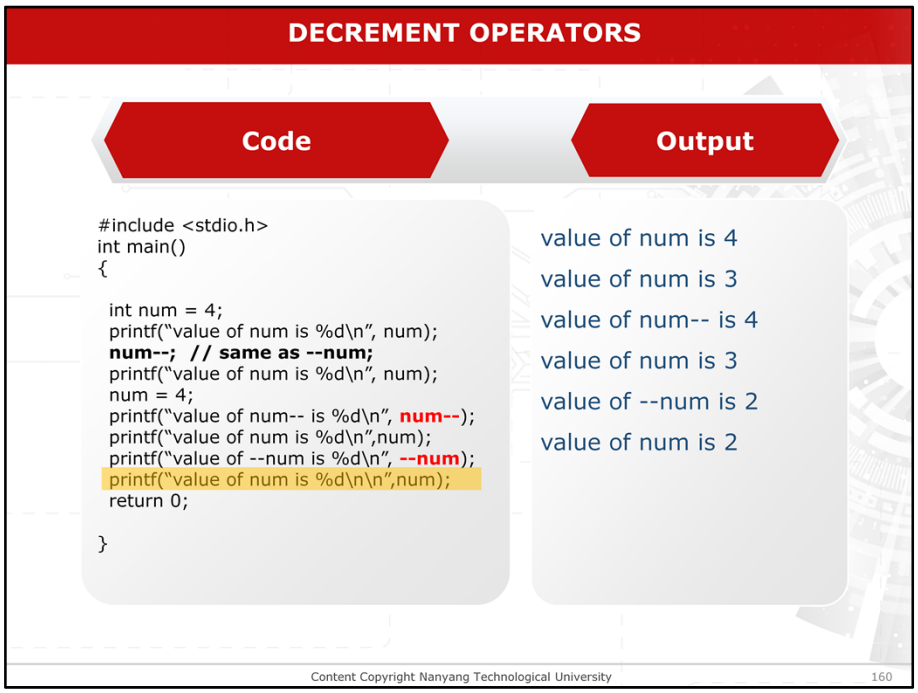
The variable **num** is assigned with the value 4 again.



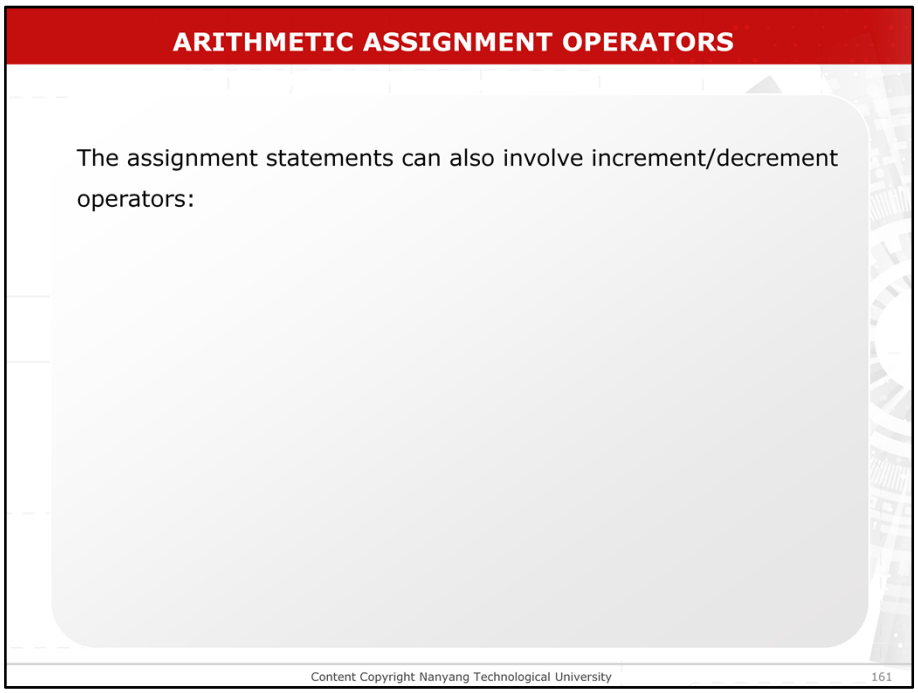
As the third **printf()** statement uses the postfix mode, the value of **num** is decremented by 1 after the printing has taken place. Therefore, the third **printf()** statement prints the value of 3 for **num**.



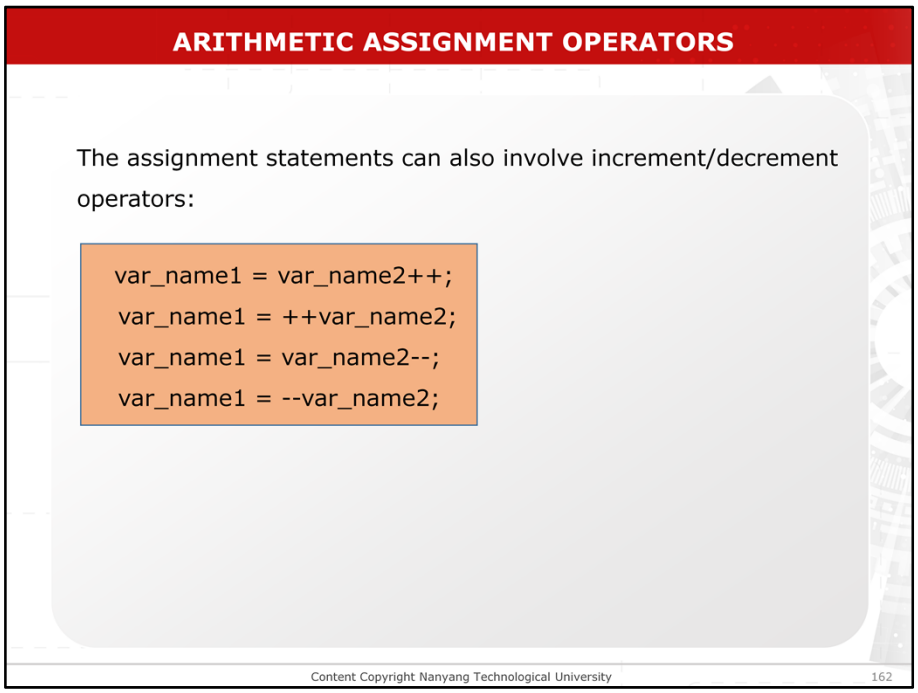
The fourth statement then prints the value of 2.



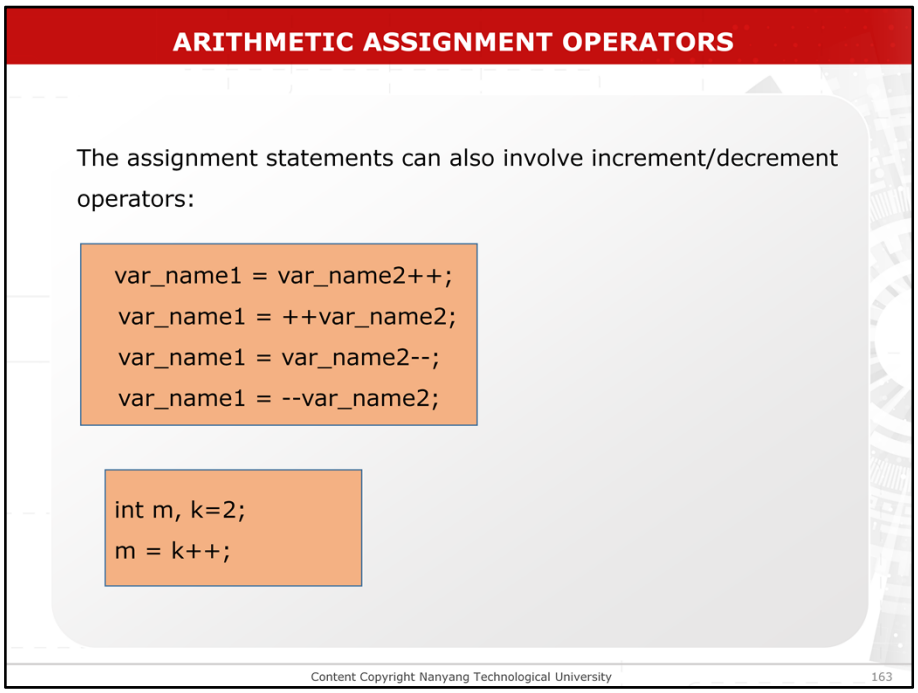
The value of **num** is decremented before the printing is taken place. Thus, it prints the value 2. The sixth statement also prints the value 2 for **num**.



The assignment statements can also involve increment/decrement operators:



as shown here.

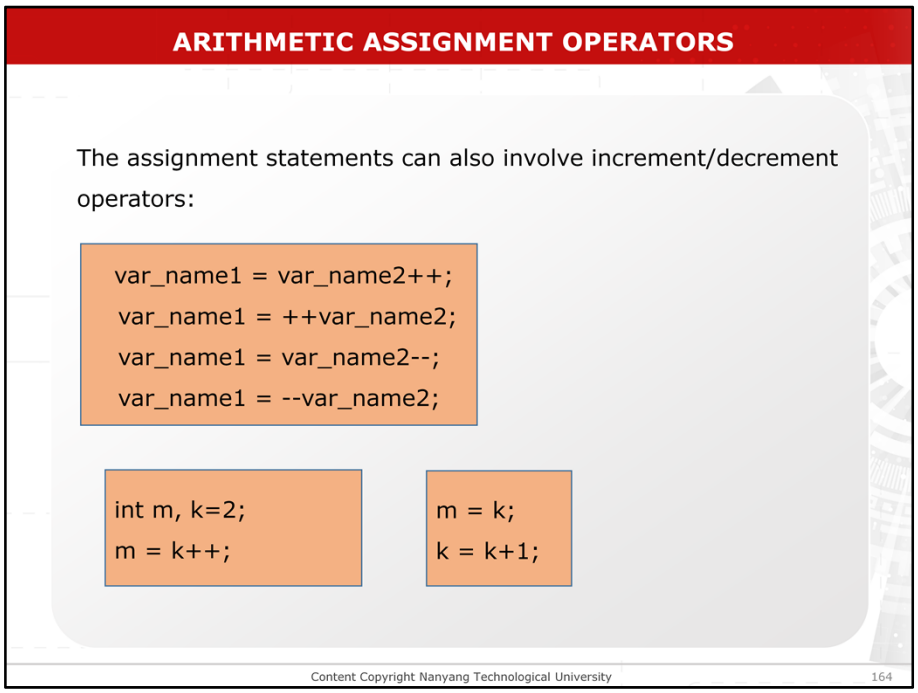


For example, if we define two variables **m** and **k**

int m, k=2;

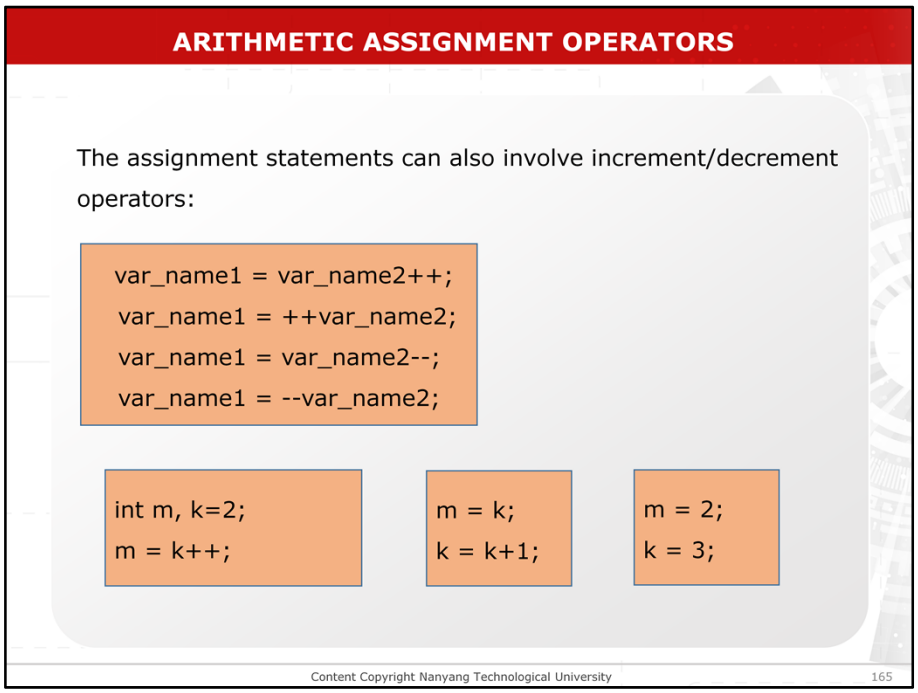
the statement

m = k++;

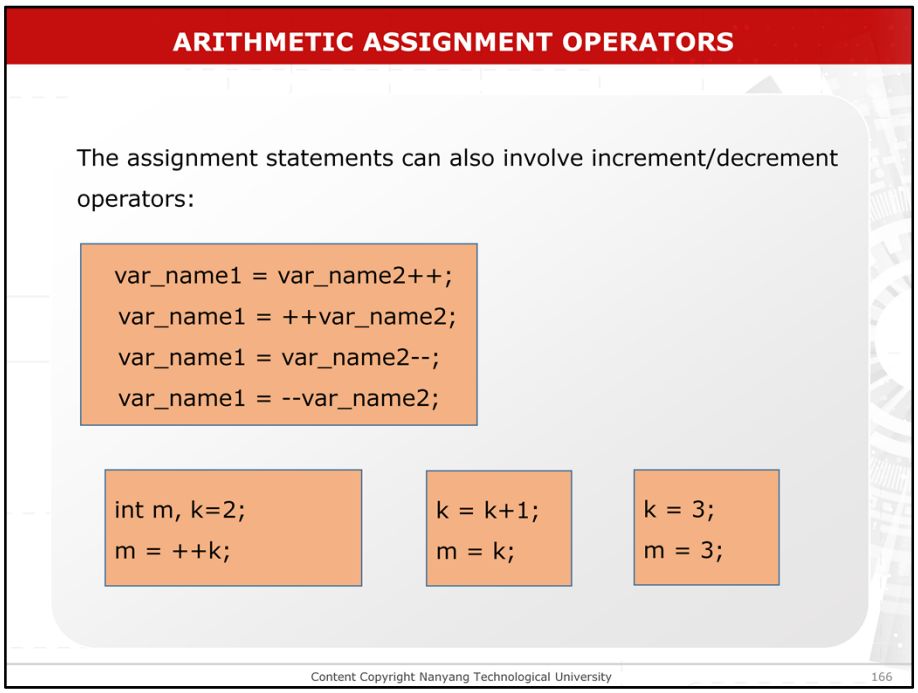


which is the same as

m = k;
k = k + 1;



will give **m** the value 2 and **k** the value 3 after executing the statement.



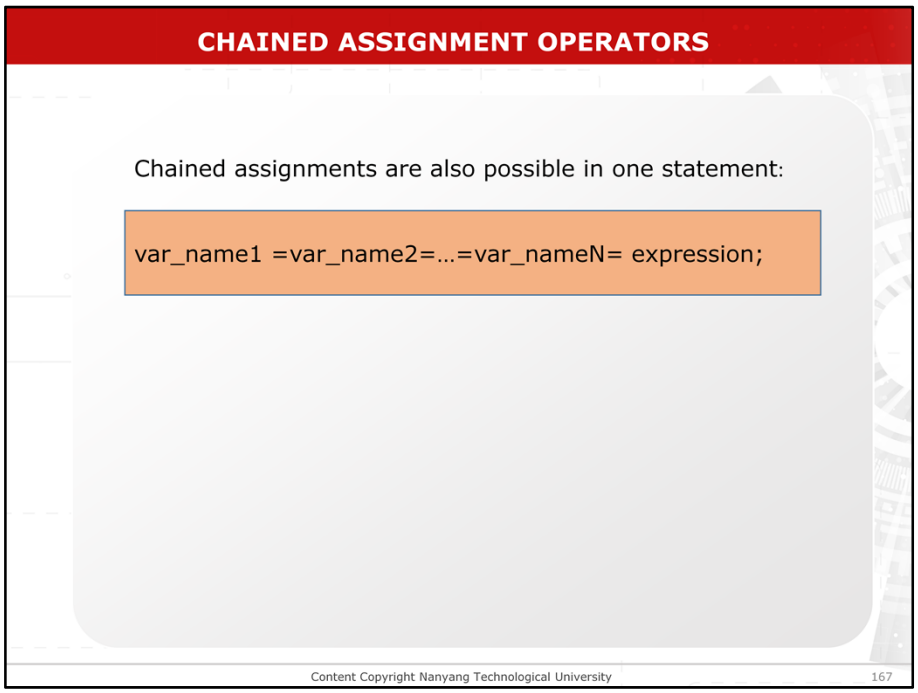
However, the statement

m = ++k;

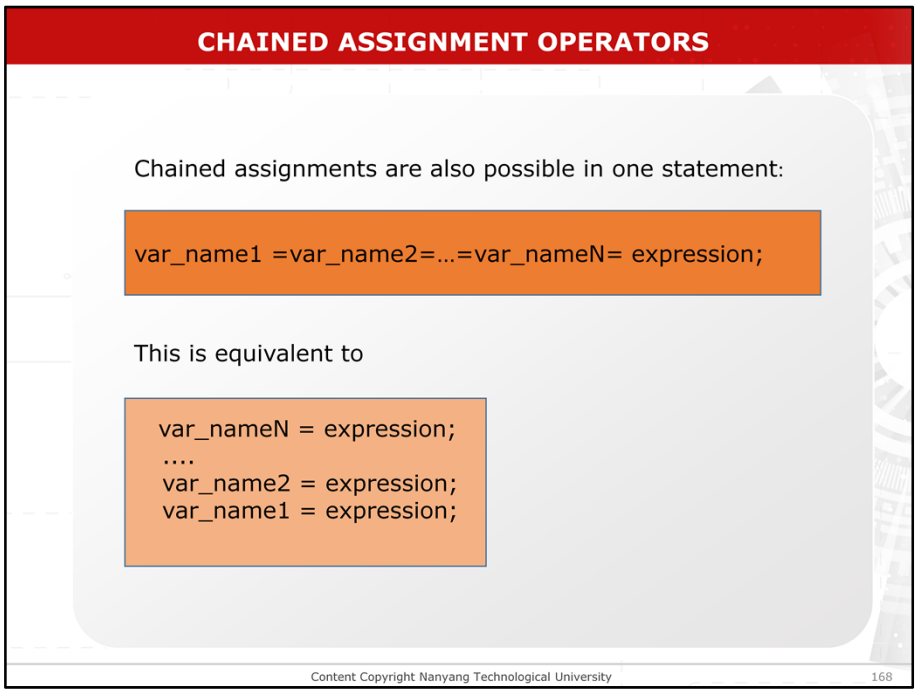
which is the same as

k = k + 1;
m = k;

will give **m** the value 3 and **k** the value 3 after executing the statement.

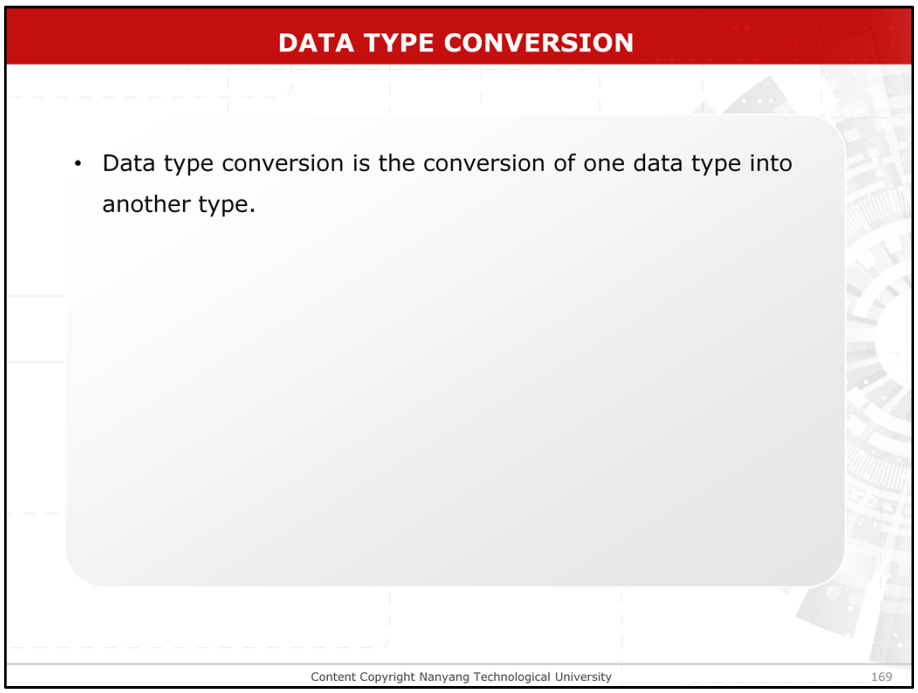


Chained Assignments
Chained assignments are also possible in one statement as shown here.



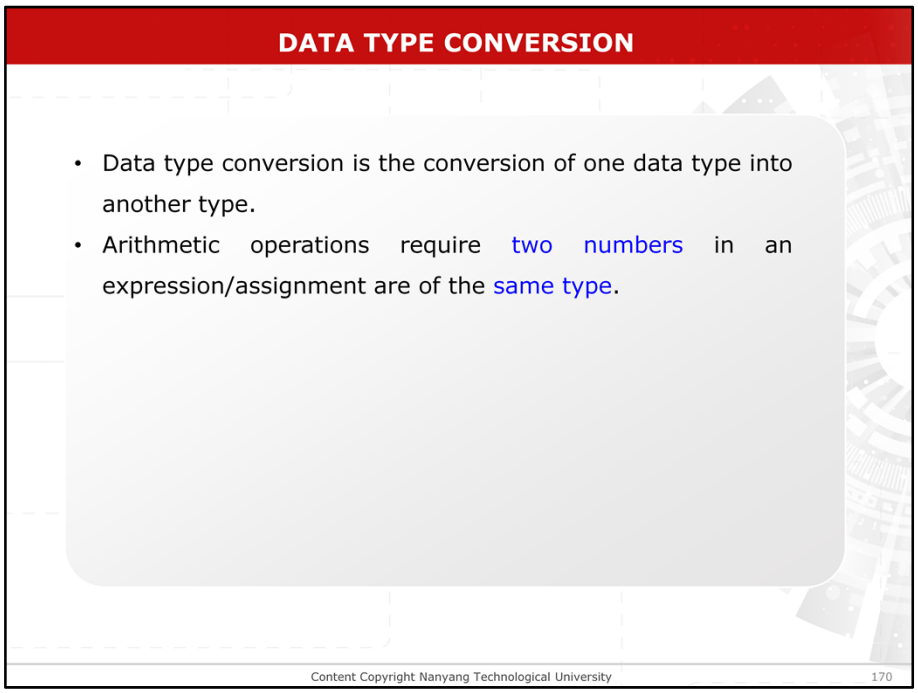
This is equivalent to

var_name2 = expression;
var_name1 = expression;

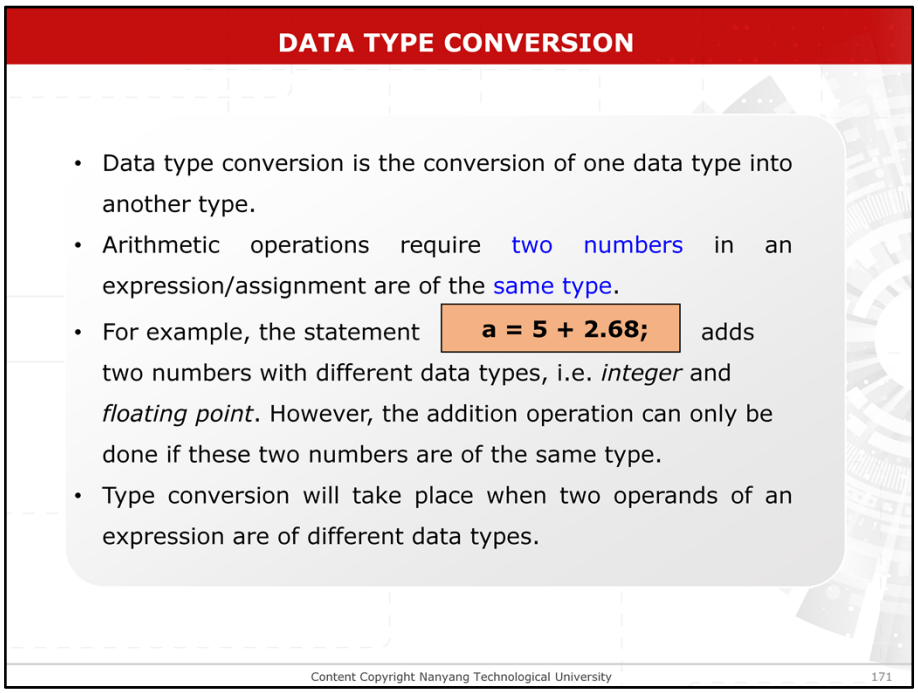


Data Type Conversion

Data type conversion is the conversion of one data type into another type.



Data type conversion is the conversion of one data type into another type. It is needed when more than one type of data objects appear in an expression or assignment.

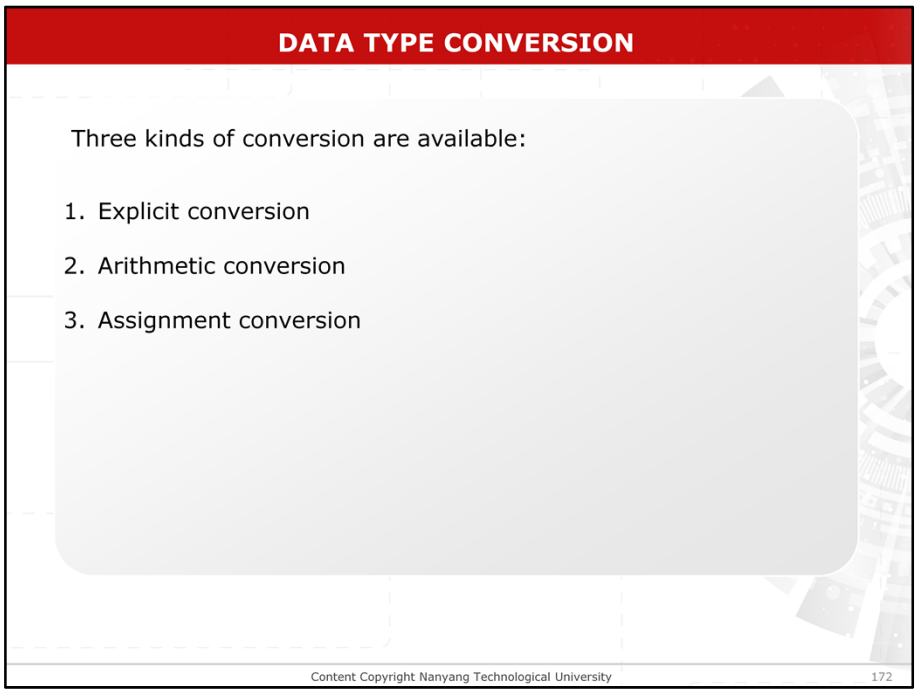


[For example, the statement

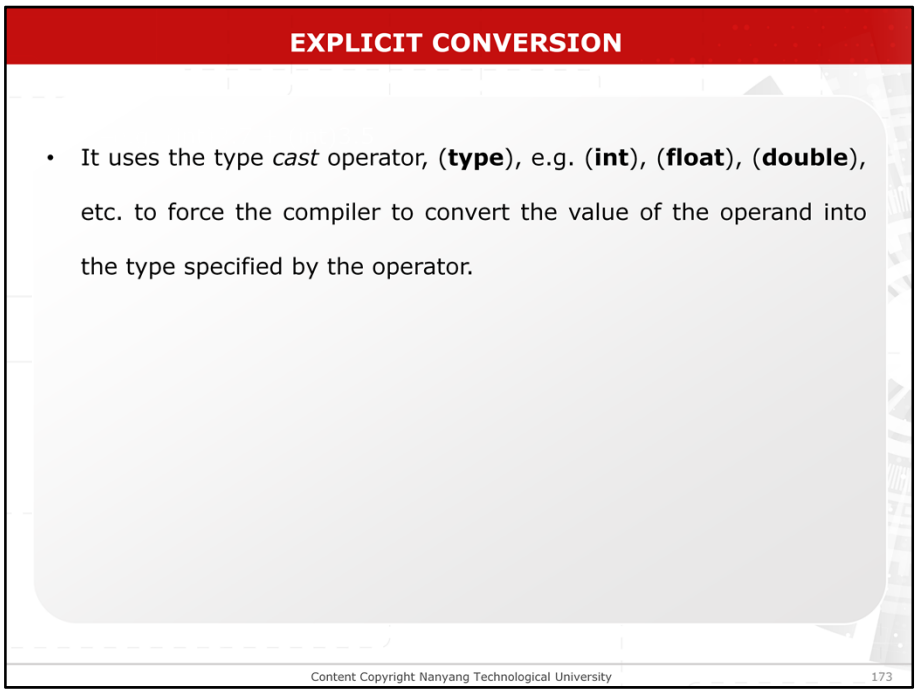
a = 5 + 2.68;

adds two numbers with different data types, i.e. integer and floating point. However, the addition operation can only be done if these two numbers are of the same type.

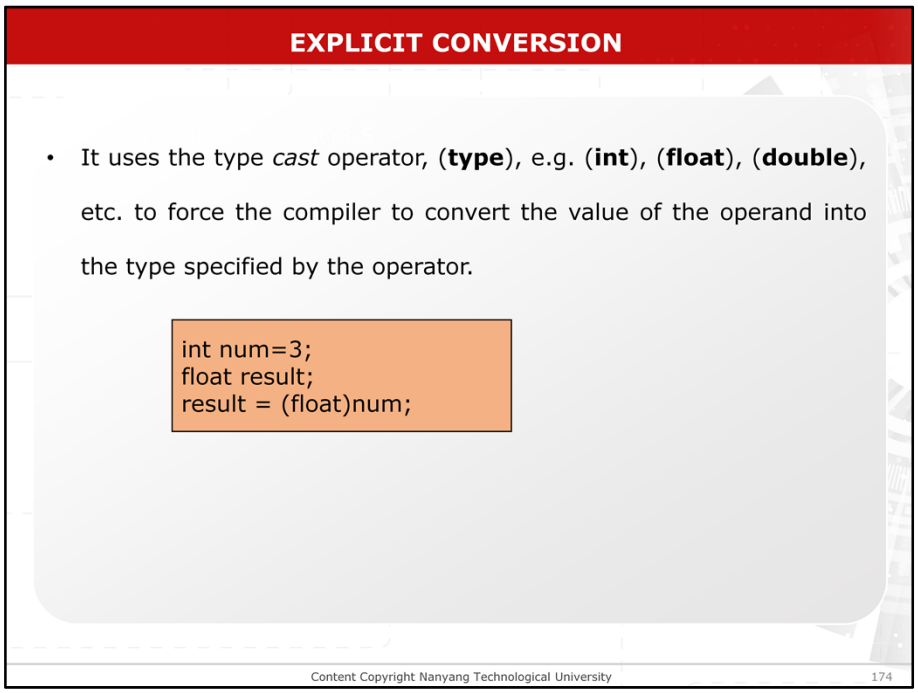
Type conversion will take place when two operands of an expression are of different data types



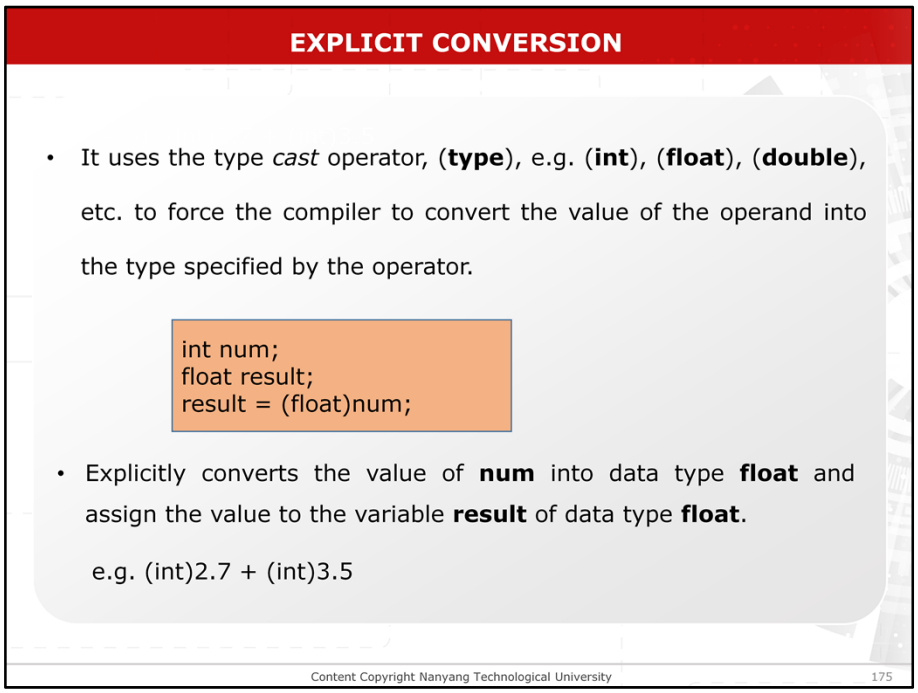
Three kinds of conversion can be performed: (1) explicit conversion; (2) arithmetic conversion; and (3) assignment conversion.



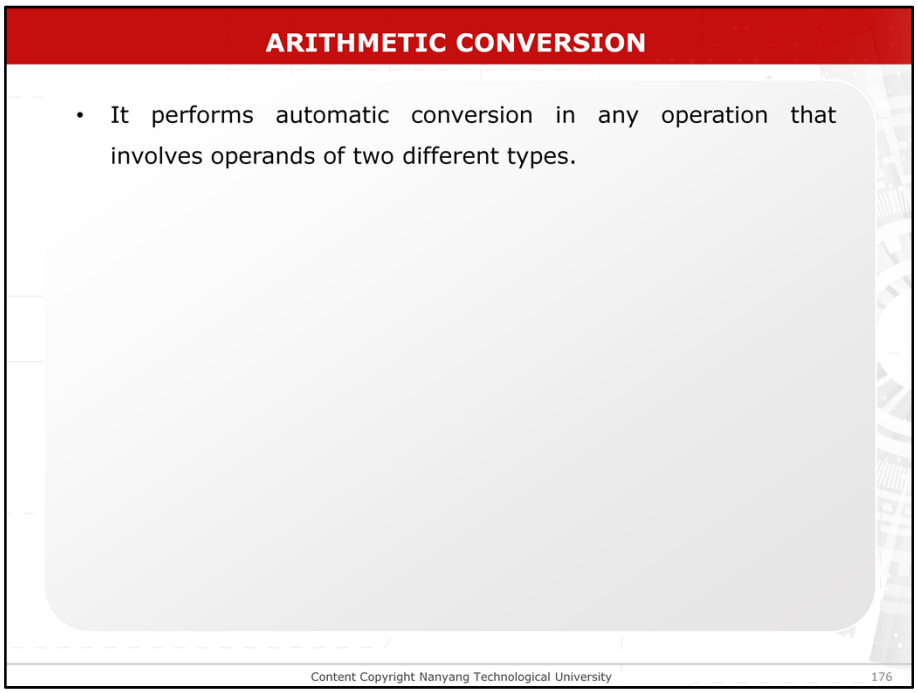
In *explicit conversion*, it uses the type *cast* operator, (**type**), e.g. (**int**), (**float**), (**double**), etc. to force the compiler to convert the value of the operand into the type specified by the operator.



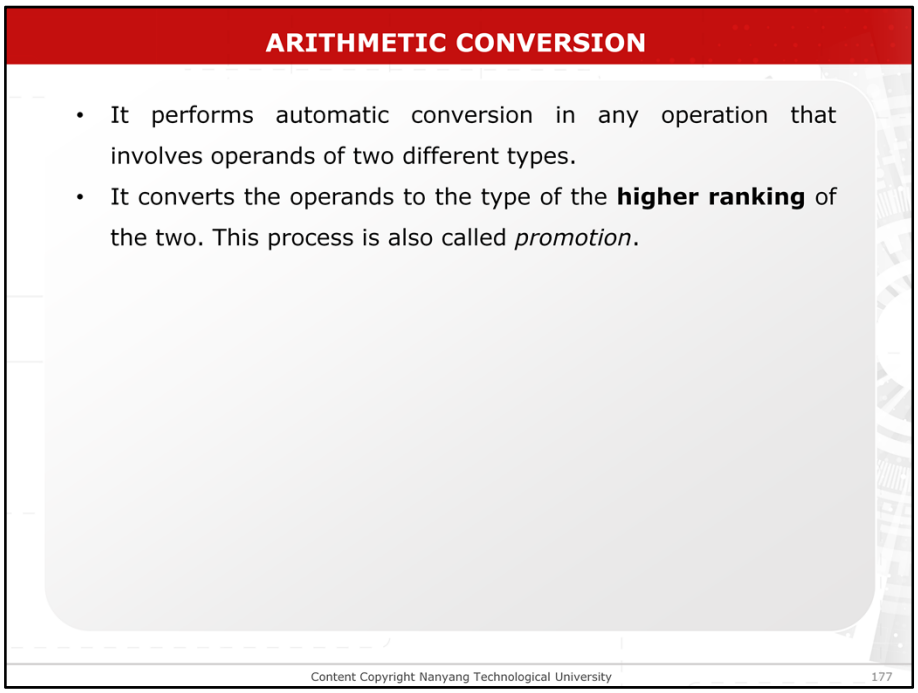
For example, the statement is shown here.



Explicitly converts the value of **num** into data type **float** and assign the value to the variable **result** of data type **float**.



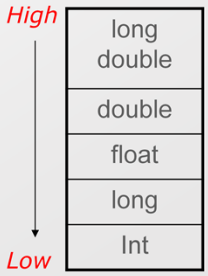
In *arithmetic conversion*, it performs automatic conversion in any operation that involves operands of two different types.



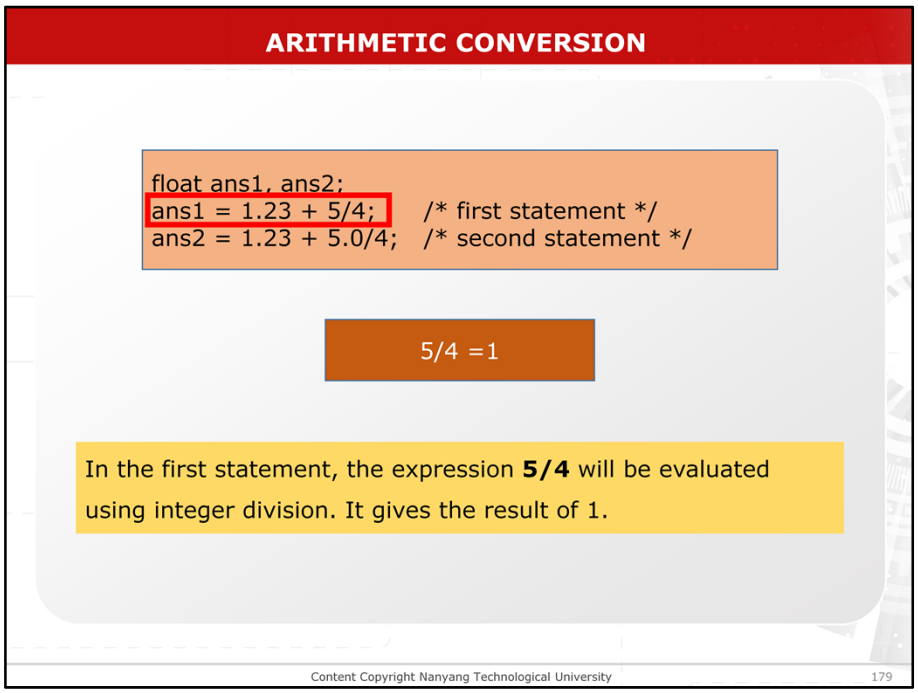
It converts the operands to the type of the higher ranking of the two. This process is also called *promotion*.

ARITHMETIC CONVERSION

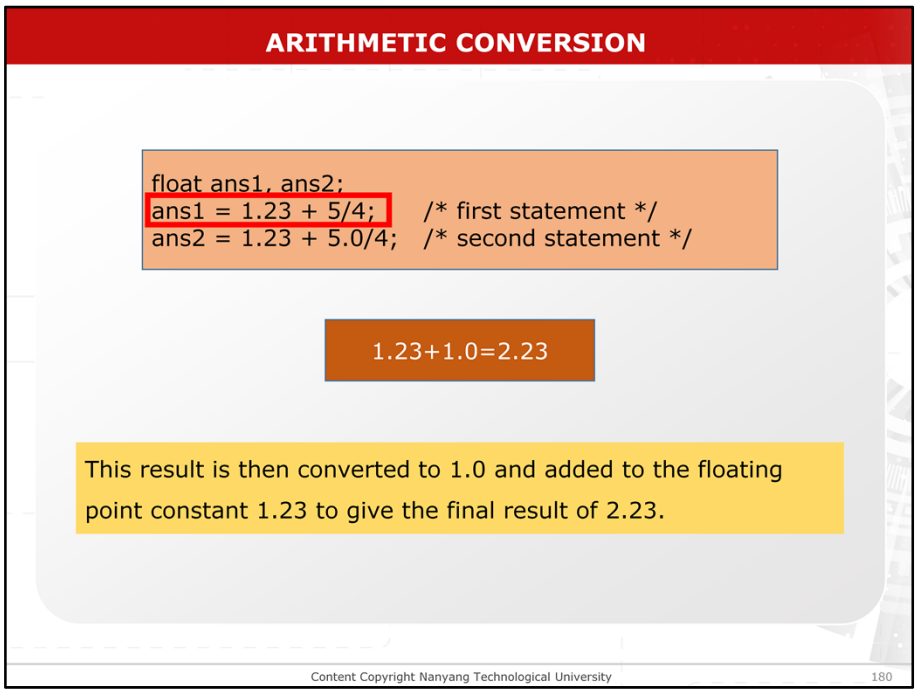
- It performs automatic conversion in any operation that involves operands of two different types.
- It converts the operands to the type of the **higher ranking** of the two. This process is also called *promotion*.
- The ranking of types is based on the amount of memory storage the value needs.



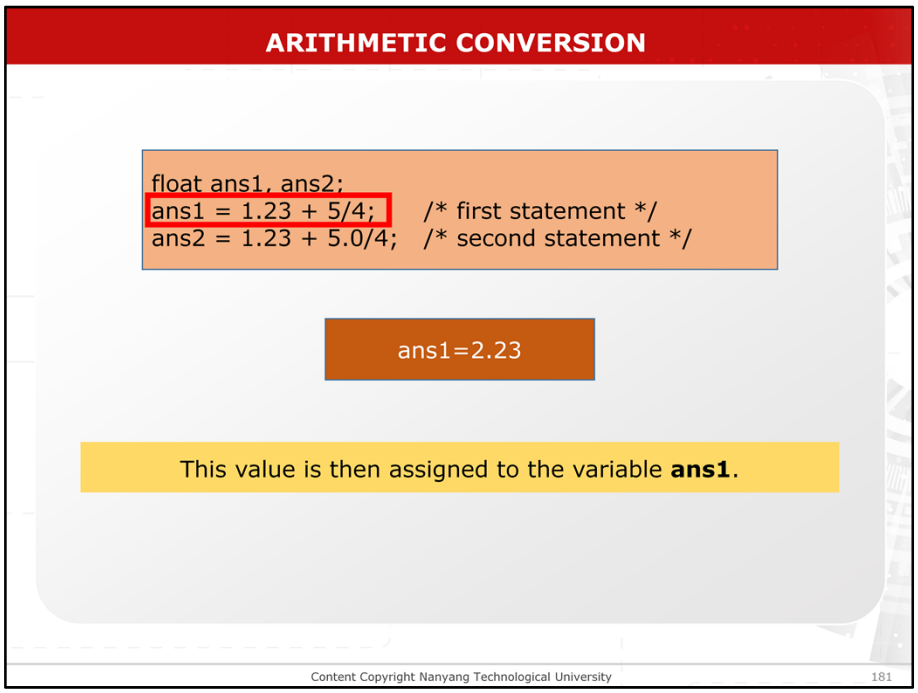
The ranking of types is based on the amount of memory storage the value needs. The ranking from high to low is as indicated by arrows.



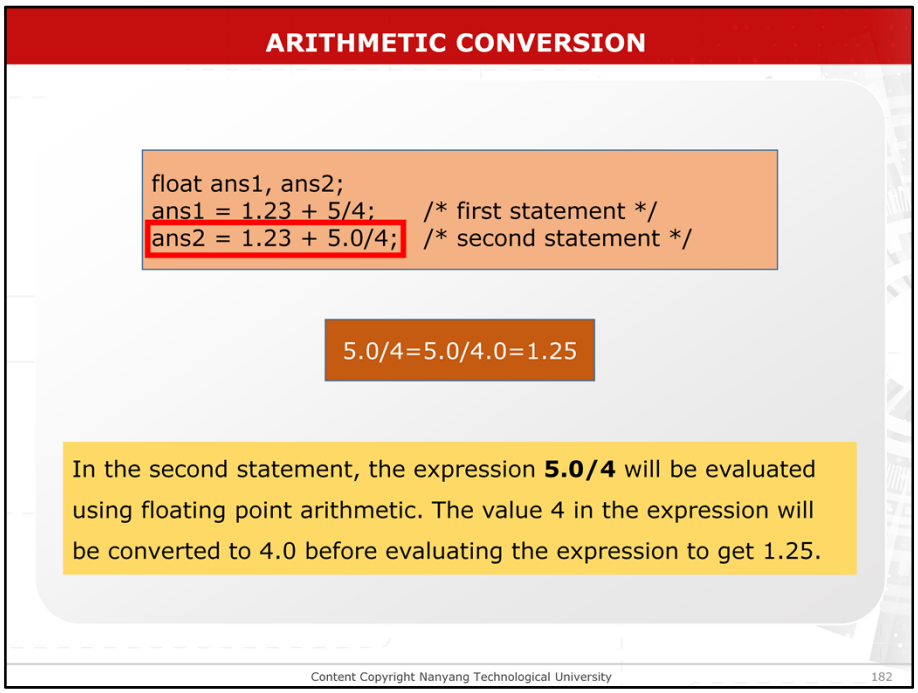
In the first statement, the expression **5/4** will be evaluated using integer division.



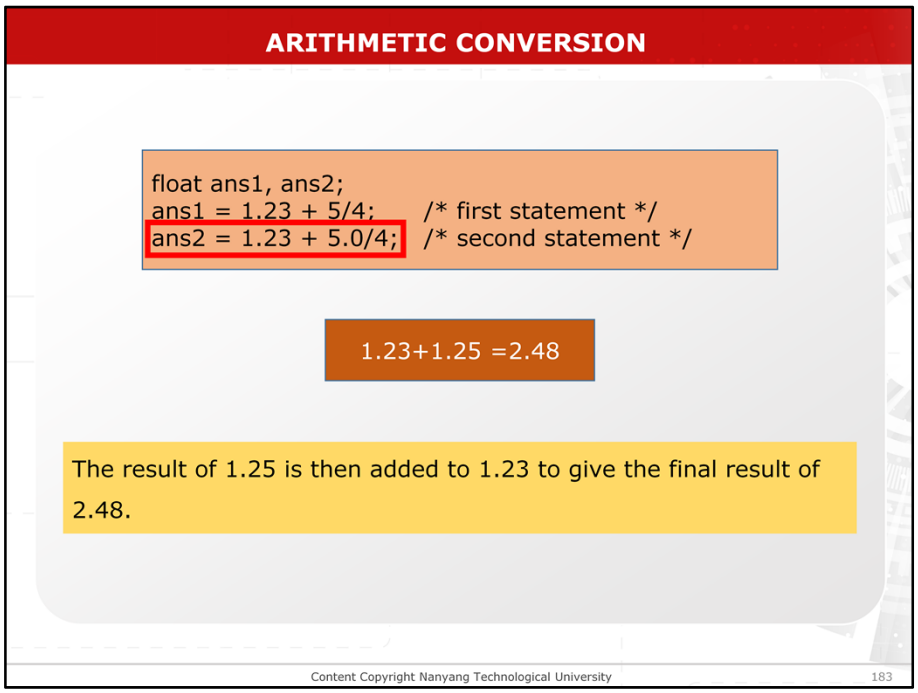
This result is then converted to 1.0 and added to the floating point constant 1.23 to give the final result of 2.23.



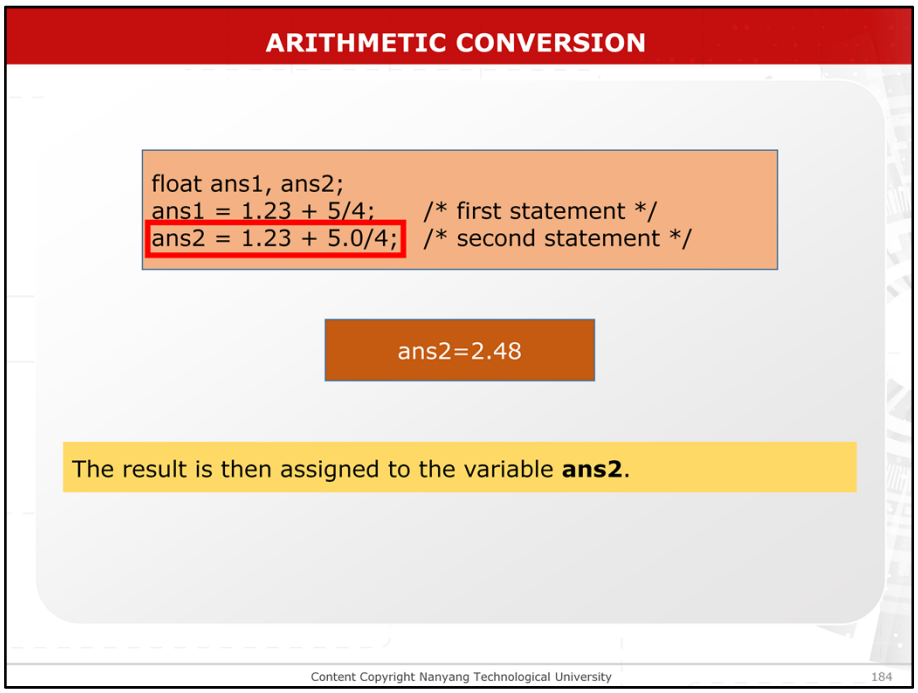
This value is then assigned to the variable **ans1**.



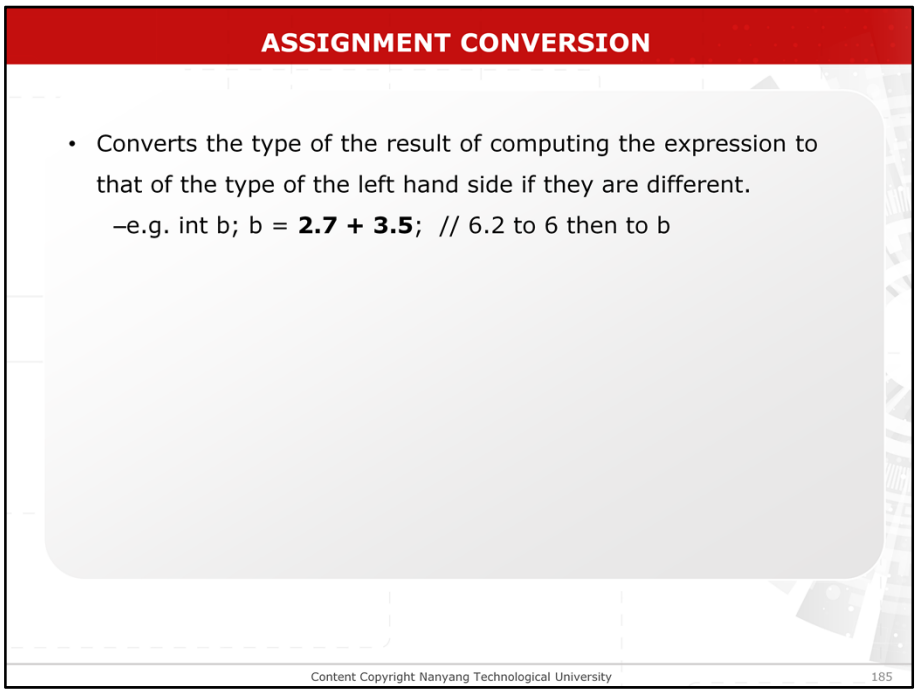
In the second statement, the expression **5.0/4** will be evaluated using floating point arithmetic. The value 4 in the expression will be converted to 4.0 before evaluating the expression to get 1.25.



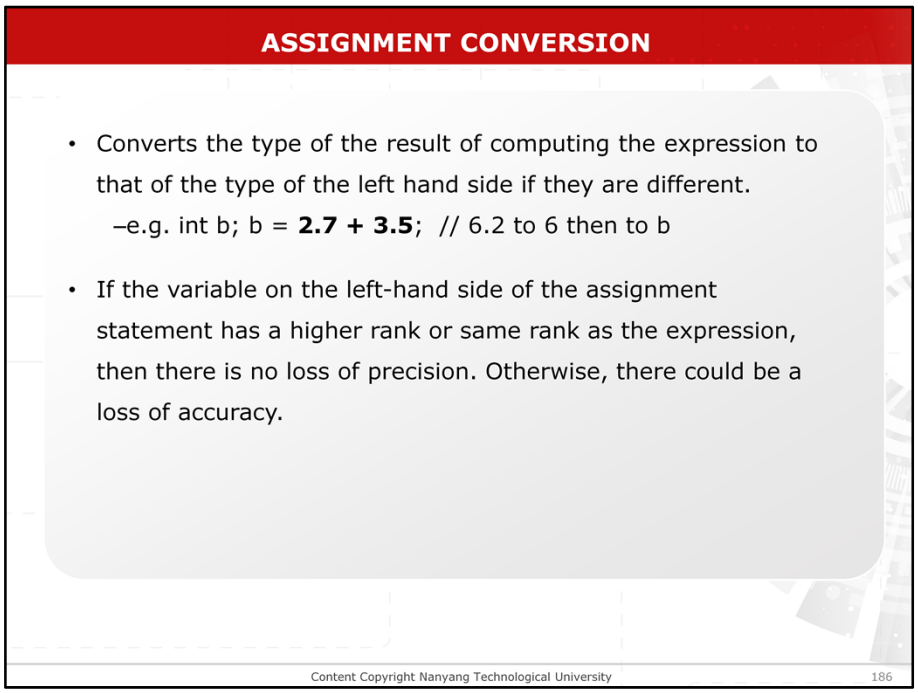
The result of 1.25 is then added to 1.23 to give the final result of 2.48.



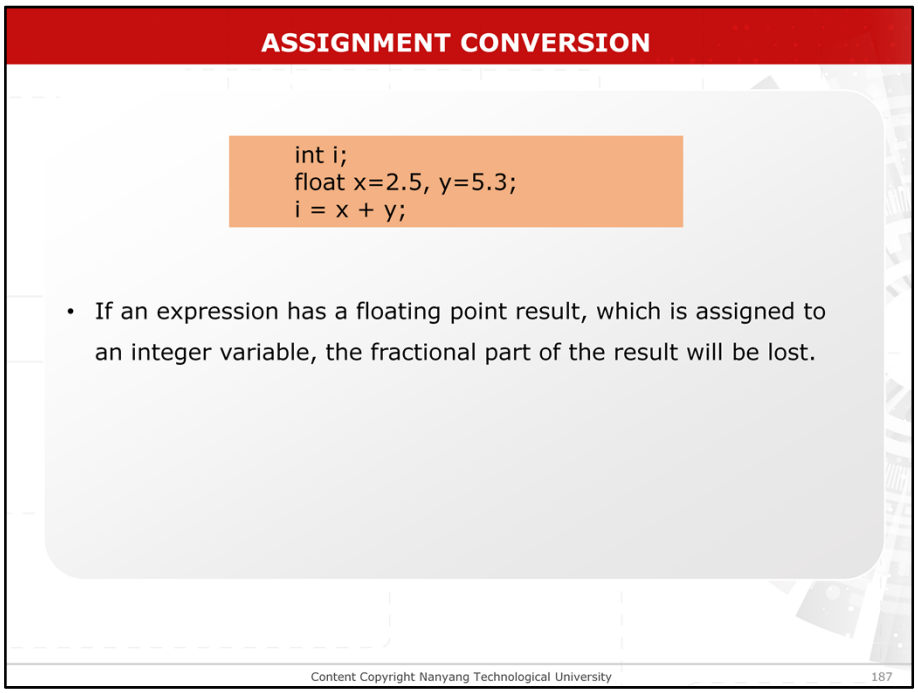
- The result is then assigned to the variable **ans2**.



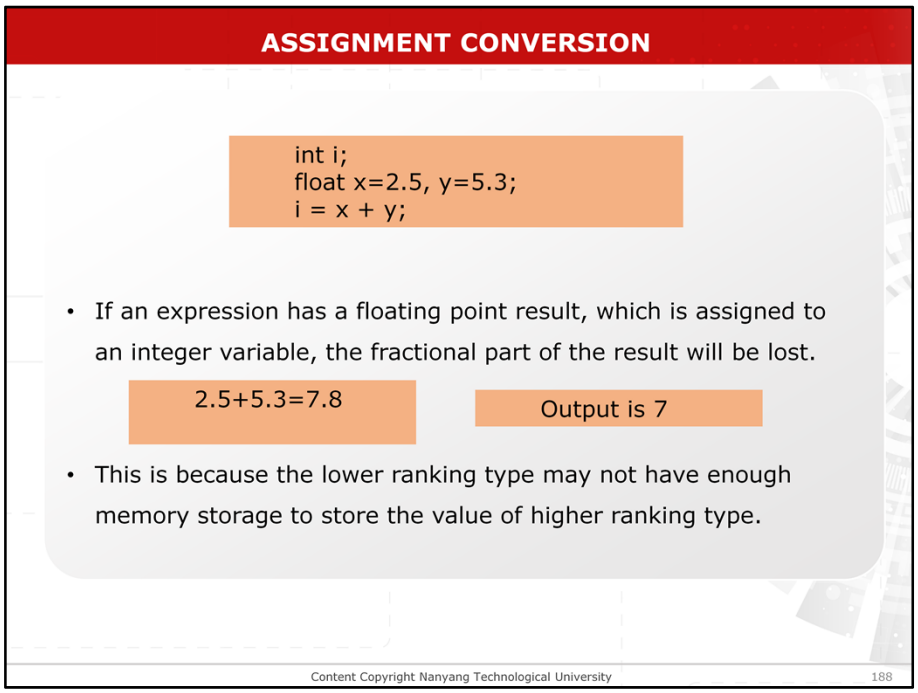
In *assignment conversion*, it converts the type of the result of computing the expression to that of the type of the left-hand side if they are different.



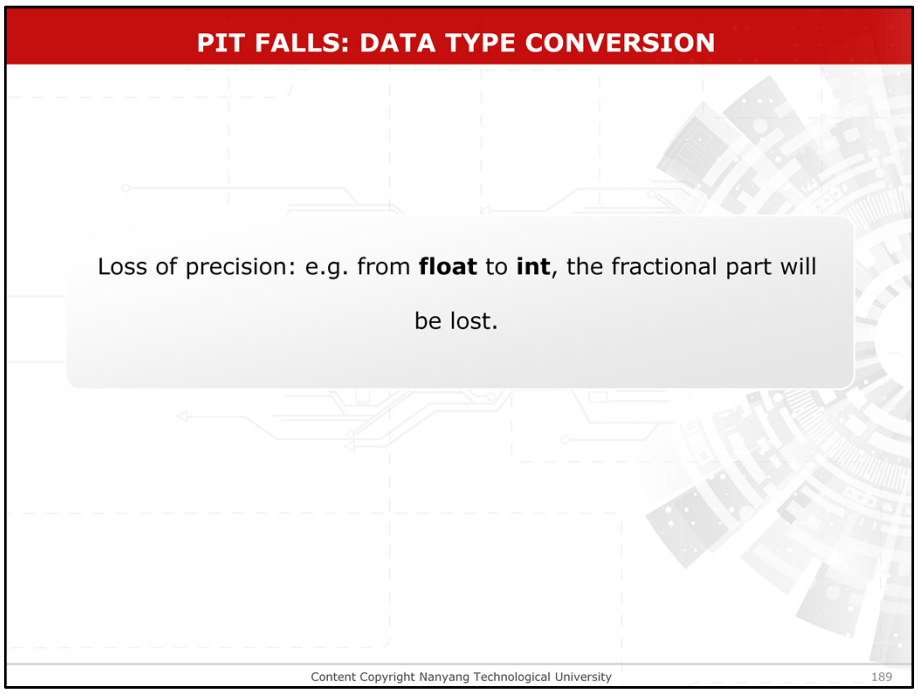
if the variable on the left-hand side of the assignment statement has a higher rank or same rank as the expression, then there is no loss of precision. Otherwise, there could be a loss of accuracy.



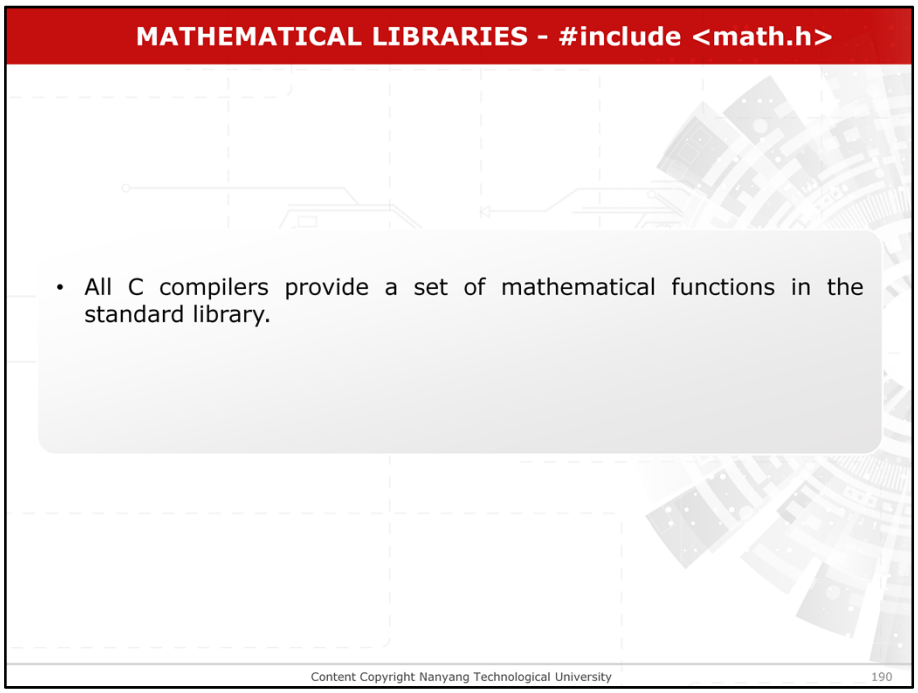
For example, if an expression has a floating point result, which is assigned to an integer variable, the fractional part of the result will be lost.



Though the value computed is 7.8, the statement will give **i** the value 7. This is because the lower ranking type may not have enough memory storage to store the value of higher ranking type.

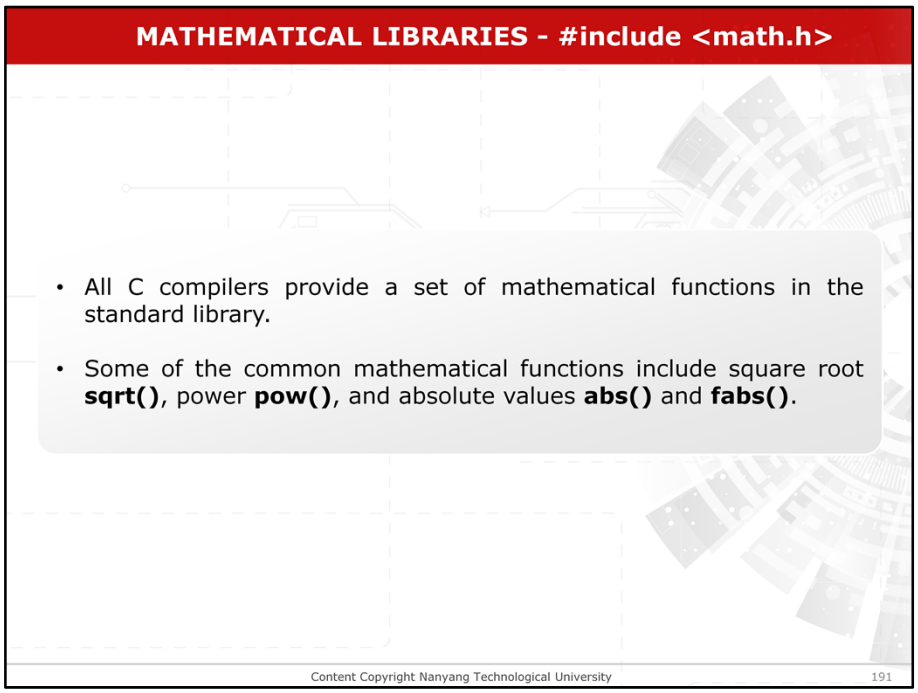


Possible *pit-falls* about type conversion is the Loss of precision, for example, when it is converted from **float** to **int**, the fractional part will be lost.



Mathematical Libraries

All C compilers provide a set of mathematical functions in the standard library.



Some of the common mathematical functions include square root **sqrt()**, power **pow()**, and absolute values **abs()** and **fabs()**.

MATHEMATICAL LIBRARIES - #include <math.h>

Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest double larger than or equal to x that can be represented as an int.	double
floor(x)	double	Return the largest double smaller than or equal to x that can be represented as an int.	double
abs(x)	int	Return the absolute value of x, where x is an int.	int
fabs(x)	double	Return the absolute value of x, where x is a floating point number.	double
sqrt(x)	double	Return the square root of x, where x >= 0.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of x, where x is in radians.	double
sin(x)	double	Return the sine of x, where x is in radians.	double
tan(x)	double	Return the tangent of x, where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x.	double
log10(x)	double	Return the base 10 logarithm of x.	double

Mathematical Libraries

The **sqrt()** function computes the square root of a value of type **double**. It returns the result of type **double**. The statement **y = sqrt(x);**
computes the square root of the value stored in the variable **x**, returns and stores the result in the variable **y**.

MATHEMATICAL LIBRARIES - #include <math.h>

Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest double larger than or equal to x that can be represented as an int.	double
floor(x)	double	Return the largest double smaller than or equal to x that can be represented as an int.	double
abs(x)	int	Return the absolute value of x, where x is an int.	int
fabs(x)	double	Return the absolute value of x, where x is a floating point number.	double
sqrt(x)	double	Return the square root of x, where x >= 0.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of x, where x is in radians.	double
sin(x)	double	Return the sine of x, where x is in radians.	double
tan(x)	double	Return the tangent of x, where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x.	double
log10(x)	double	Return the base 10 logarithm of x.	double

The functions **abs()** and **fabs()** give the absolute values of an integer and a floating point number respectively.

MATHEMATICAL LIBRARIES - #include <math.h>

Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest double larger than or equal to x that can be represented as an int.	double
floor(x)	double	Return the largest double smaller than or equal to x that can be represented as an int.	double
abs(x)	int	Return the absolute value of x, where x is an int.	int
fabs(x)	double	Return the absolute value of x, where x is a floating point number.	double
sqrt(x)	double	Return the square root of x, where x >= 0.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of x, where x is in radians.	double
sin(x)	double	Return the sine of x, where x is in radians.	double
tan(x)	double	Return the tangent of x, where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x.	double
log10(x)	double	Return the base 10 logarithm of x.	double

The **pow()** function will take in two arguments, **x** and **y**, and returns $\mathbf{x^y}$.

MATHEMATICAL LIBRARIES - #include <math.h>

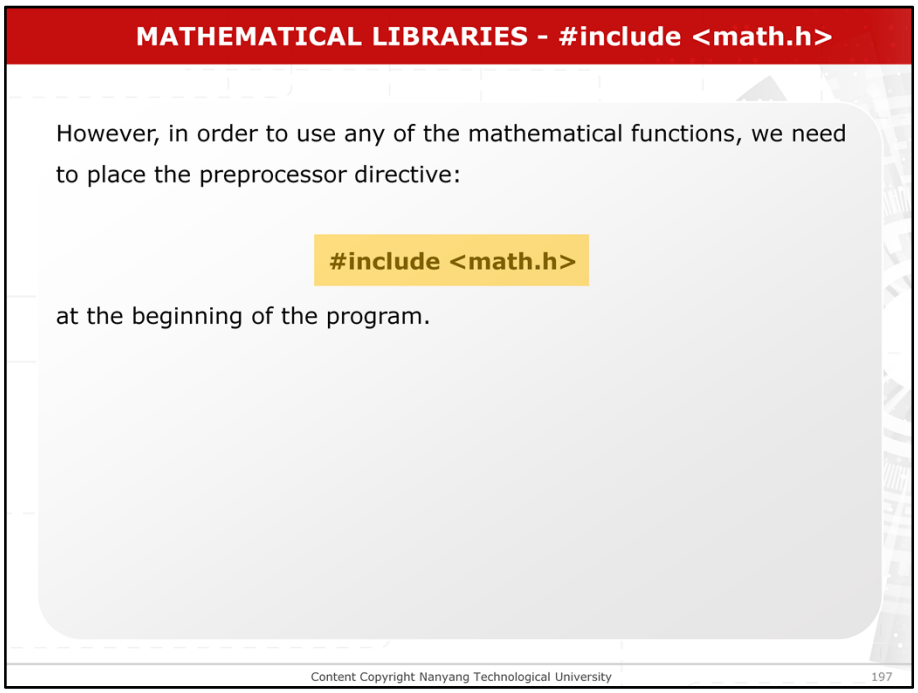
Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest double larger than or equal to x that can be represented as an int.	double
floor(x)	double	Return the largest double smaller than or equal to x that can be represented as an int.	double
abs(x)	int	Return the absolute value of x, where x is an int.	int
fabs(x)	double	Return the absolute value of x, where x is a floating point number.	double
sqrt(x)	double	Return the square root of x, where x >= 0.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of x, where x is in radians.	double
sin(x)	double	Return the sine of x, where x is in radians.	double
tan(x)	double	Return the tangent of x, where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x.	double
log10(x)	double	Return the base 10 logarithm of x.	double

Apart from the above functions, trigonometric functions such as **sin()**, **cos()** and **tan()** that compute the sine, cosine and tangent of an angle are also provided.

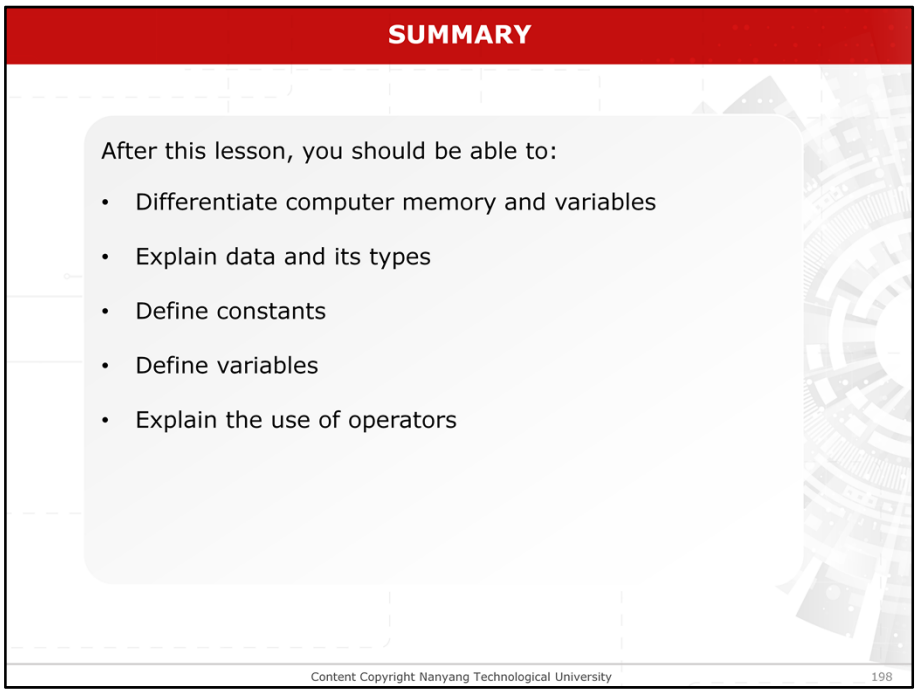
MATHEMATICAL LIBRARIES - #include <math.h>

Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest double larger than or equal to x that can be represented as an int.	double
floor(x)	double	Return the largest double smaller than or equal to x that can be represented as an int.	double
abs(x)	int	Return the absolute value of x, where x is an int.	int
fabs(x)	double	Return the absolute value of x, where x is a floating point number.	double
sqrt(x)	double	Return the square root of x, where x >= 0.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of x, where x is in radians.	double
sin(x)	double	Return the sine of x, where x is in radians.	double
tan(x)	double	Return the tangent of x, where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x.	double
log10(x)	double	Return the base 10 logarithm of x.	double

Apart from the above functions, trigonometric functions such as **sine()**, **cos()** and **tan()** that compute the sine, cosine and tangent of an angle are also provided.



However, in order to use any of the mathematical functions, we need to place the preprocessor directive
at the beginning of the program.



After watching the video lesson, you should be able to do the listed.