

SC1008 C and C++ Programming

Assistant Professor WANG Yong

yong-wang@ntu.edu.sg

CCDS, Nanyang Technological University

Week 8

C++ Programming Basics

Why Learning C++?

- C++ is one of the most popular programming languages








[About us](#) [Knowledge](#) [News](#) [Coding Standards](#) [TIOBE Index](#) [Contact](#) [🔍](#)

[Products](#) [Quality Models](#) [Markets](#) [Schedule a demo](#)

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular web sites Google, Amazon, Wikipedia, Bing and more than 20 others are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found [here](#).

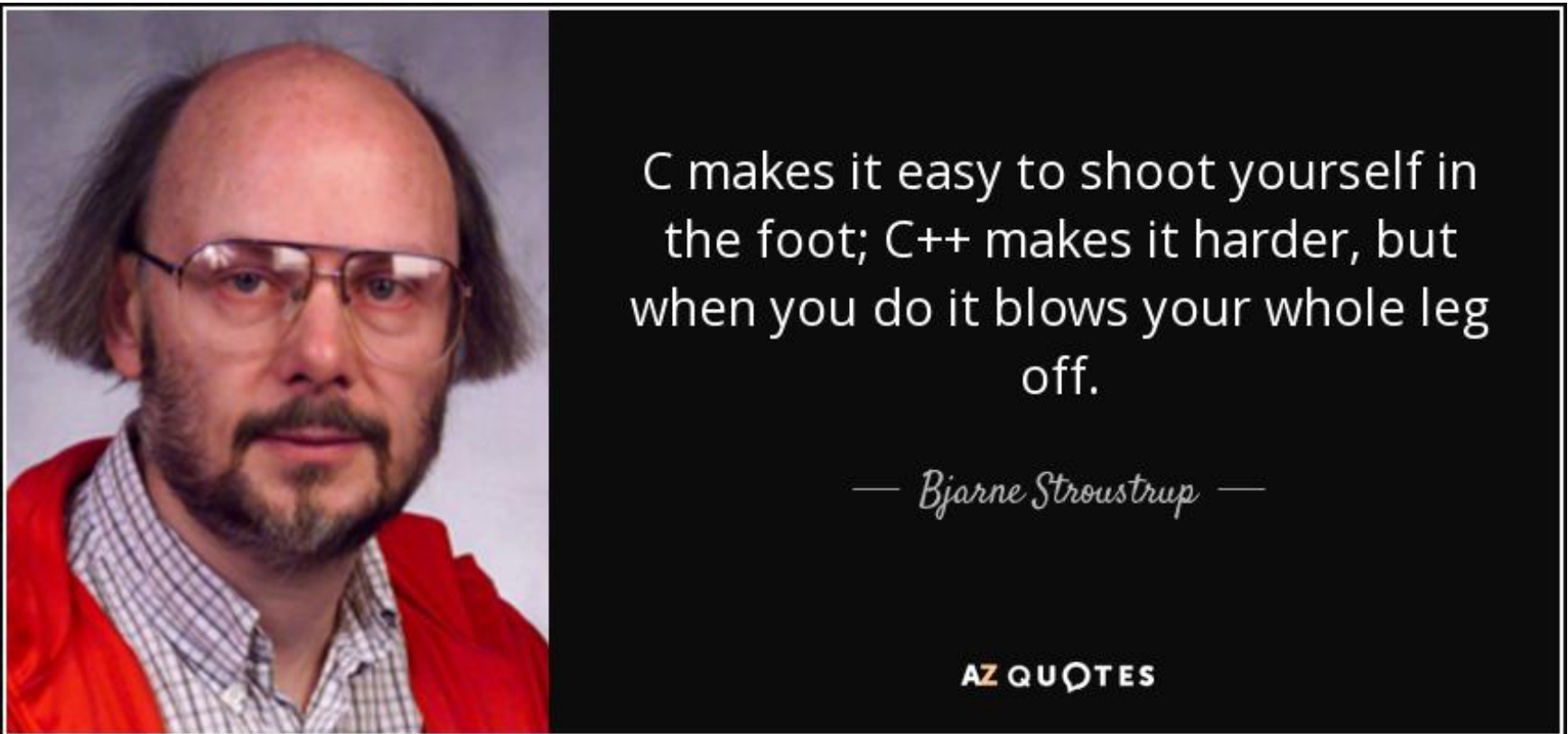
Feb 2025	Feb 2024	Change	Programming Language		Ratings	Change
1	1			Python	23.88%	+8.72%
2	3	↑		C++	11.37%	+0.84%
3	4	↑		Java	10.66%	+1.79%
4	2	↓		C	9.84%	-1.14%
5	5			C#	4.12%	-3.41%

- Many reasons to learn C++

Popularity and high salary; abundant library support; large community; being portable; wide usage in databases, OS, graphics, etc.

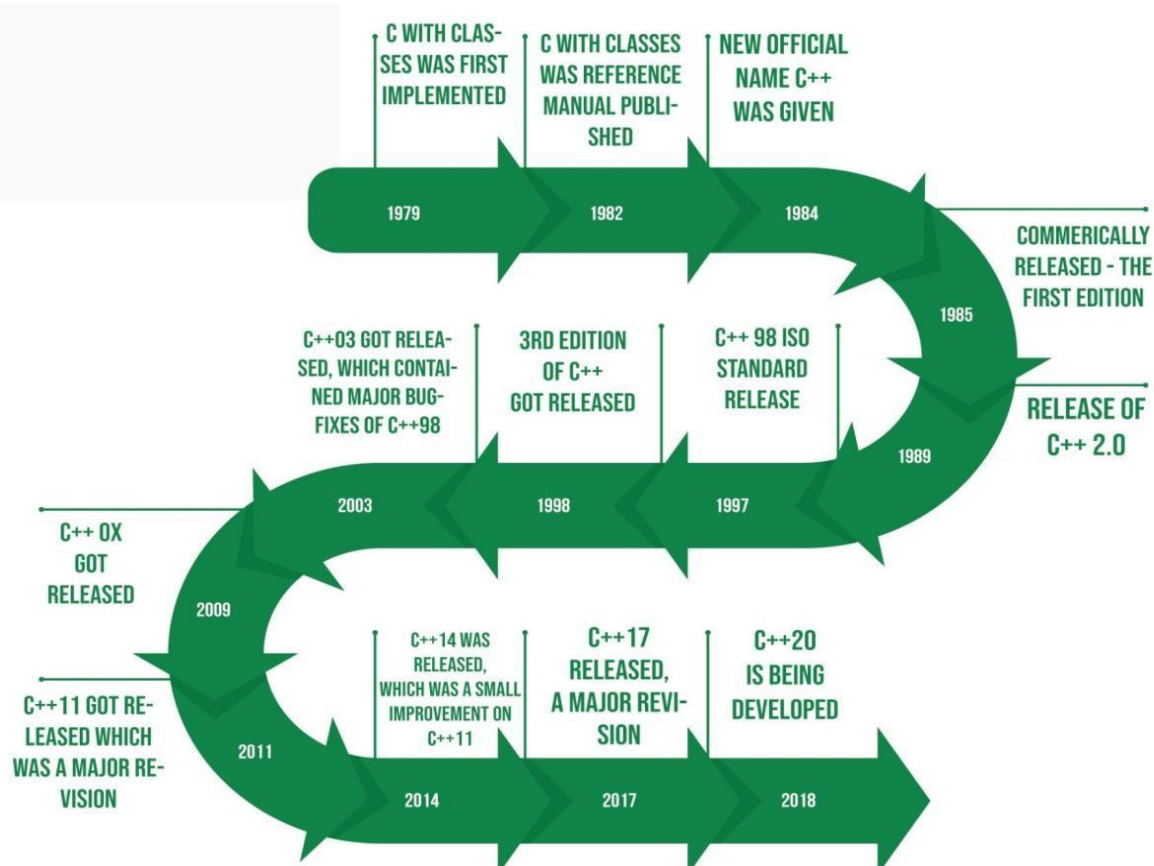
What is C++?

- C++ is a general-purpose, object-oriented programming language that was designed by *Bjarne Stroustrup* in 1979 to be an extension of the C language.



What is C++?

History of C++



What is C++?

- Developed by Bjarne Stroustrup, as an extension to the C language (1979).
- Danish Computer Scientist at Bell Telephone Laboratories (Nokia Bell Labs) in New Jersey
- Simula (Used for simulations) : primary language to support the object-oriented programming- too slow for practice & practical use.
- Worked on “**C with Classes**” to get advanced object-oriented programming, into
 - the C language
- In 1983, the name Change from C with categories to C++.
 - The ++ operator for incrementing a variable
- The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

What is C++?

- C++ vs. C:
 - C++ is a superset of C, meaning that any valid C program is also a valid C++ program
 - C++ joins three separate programming categories:
 - Procedural language, represented by C;
 - *Object-oriented programming (OOP) language*, represented by the class enhancements in C++;
 - *Generic programming*, supported by C++ templates.
- In the second half of this course, we emphasize more on OOP and generic programming

OOP heritage provides
a high level of abstraction.



```
....  
north_america.show();  
....
```

C heritage provides
low-level hardware access.



```
set byte at  
address  
01000 to 0
```

- Input/output
- Data types:
 - Basic data types
 - Compound data types
- Reference variables
- Dynamic memory allocation
- Functions
 - Inline functions
 - Default arguments
 - Function overloading
 - Function templates

Input/Output: Your First C++ Program

Note: c++ program files often end with an extension of ".cpp" (or ".cc" or ".cxx")


```
//main.cpp
#include <iostream> //header of input/output stream
using namespace std; //namespace of standard library
int main()
{
    //cout: console output stream object
    //endl: start a new line. Its effect is equivalent to '\n' in
    C program
        cout << "Hello world!" << endl;
        cout << "I'm from NTU.\n";
        return 0;
}
```

```
//Output in the display/terminal:
Hello world!
I'm from NTU.
```

- Namespaces:
 - A new feature introduced in C++
 - Designed to avoid possible name conflicts of multiple libraries, e.g., two libraries may both define classes (or functions, variables) named `List`, `Tree`, and `Node`, how can we differentiate them?
 - `std` namespace is defined in the `iostream` file and contains the definitions of `cin` and `cout`, etc.

```
#include <iostream>
//using namespace std;
int main()
{
    std::cout << "Hello world!" << std::endl;
    std::cout << "I'm from NTU.\n";
    return 0;
}
```

```
std::cout << "Hello world!";
std::cout << std::endl;
```



If we do not use the namespace directive, we need to keep the `std::` prefix for the functions/variables/objects within the `std` namespace.

- Namespaces:

```
// In "greet.h"
#pragma once
#include <iostream>

namespace English {
    void greet() {
        std::cout << "Hello!" << std::endl;
    }
}

namespace Spanish {
    void greet() {
        std::cout << "¡Hola!" << std::endl;
    }
}
```

```
#include <iostream>
#include "greet.h"
using namespace English;

int main() {
    // Calls the function from the English namespace
    greet();

    // Calls the function from the Spanish namespace
    Spanish::greet();

    return 0;
}
```

Input/Output

- Output different variable types

Output:

```
Age: 25
Value of Pi: 3.14159
Large number: 1.23457e+06
Your grade: A
Hello, Alice!
Is student? true
a: 10, b: 20
Default: 123.457
Fixed: 123.46
```

```
#include <iostream>
#include <iomanip> // For setprecision and fixed
using namespace std;

int main() {
    int age = 25;
    cout << "Age: " << age << endl;

    float pi = 3.14159;
    cout << "Value of Pi: " << pi << endl;

    double largeNumber = 1234567.89;
    cout << "Large number: " << largeNumber << endl;

    char grade = 'A';
    cout << "Your grade: " << grade << endl;

    char name[] = "Alice"; // Character array
    cout << "Hello, " << name << "!" << endl;

    bool isStudent = true; // Boolean
    cout << "Is student? " << boolalpha << isStudent << endl;

    // Multiple variables in one line
    int a = 10, b = 20;
    cout << "a: " << a << ", b: " << b << endl;

    // Formatted floating-point output
    double num = 123.456789;
    cout << "Default: " << num << endl;
    cout << "Fixed: " << fixed << setprecision(2) << num << endl;

    return 0;
}
```

Input/Output – Handle String and Number

- Using `cin`:

```
#include <iostream>
using namespace std;
int main()
{
    int age = 0;
    char name[20];
    cout << "What is your name?" << endl;

    cin >> name;

    cout << "What is your age?" << endl;
    cin >> age;

    cout << "Your name is " << name << ", and your age is " << age << "." << endl;
    return 0;
}
```

Input stream:

Y	o	n	g		W	A	N	G	\n
---	---	---	---	--	---	---	---	---	----

Input stream:

	W	A	N	G	\n
--	---	---	---	---	----

name: "Yong"

Failed! Cannot extract a valid part of digits from the input stream!

What is your name?

Yong

What is your age?

18

Your name is Yong, and your age is 18.



What is your name?

Yong WANG

What is your age?

Your name is Yong, and your age is 0.



You don't even have a chance to enter

the age!

The `cin` will stop at whitespaces like *spaces*, *tabs* and *newlines*.

Input/Output – Handle Number

- What will be the result of `num` for different user inputs?

```
int num = 0;  
cin >> num;
```

Input stream:

	2	3	4		W	A	N	G	\n
--	---	---	---	--	---	---	---	---	----

`num: 234`

Input stream:

9	8	A	N	G	\n
---	---	---	---	---	----

`num: 98`

Input stream:

\n		2	.	3	\n
----	--	---	---	---	----

`num: 2`

Input stream:

\n	w	2	3	G	\n
----	---	---	---	---	----

Failed! Cannot extract a valid part of digits from the input stream!

1. `cin` will extract as much as it can to match the data type, until it reach white spaces or other non-digit characters.

2. After `cin` extracts the required value, any remaining characters will stay in the input stream/buffer.

```
float num = 0;  
cin >> num;
```

Input stream:

1	2	.	9	2	\n
---	---	---	---	---	----

`num: 12.92`

Input stream:

	2	.	9	2	\n
--	---	---	---	---	----

`num: 2.92`

Input stream:

9	2	.		2	\n
---	---	---	--	---	----

`num: 92`

Input stream:

\n	2	.		2	\n
----	---	---	--	---	----

`num: 2`

Input stream:

.	2	4	.	2	\n
---	---	---	---	---	----

`num: 0.24`

Input stream:

y	2	4		2	\n
---	---	---	--	---	----

Failed! Cannot extract a valid part of digits from the input stream!

Input/Output – Handle Number

- How should we cope with the failure when using `cin`?

```
int num = 0;  
cin >> num;
```

```
float num = 0;  
cin >> num;
```

Input stream:

\n	w	2	3	G	\n
----	---	---	---	---	----

Failed! Cannot extract a valid part of digits from the input stream!

Input stream:

y	2	4		2	\n
---	---	---	--	---	----

Failed! Cannot extract a valid part of digits from the input stream!

- If `cin` failed, we need to remove the invalid input from the input stream/buffer by using `cin.ignore()`;
- Clear the error state using `cin.clear()`;
- Ask the user to enter valid input again.

```
if (cin.fail()) { // Check if input failed  
  
    cin.clear(); // Clear error state  
    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard invalid input  
    cout << "Invalid input! Please enter a valid integer.\n";  
  
}
```

Input/Output – Handle String

- `cin.getline()` and `cin.get()`
 - `cin` is essentially an object of class `istream`, so it can have associated functions
 - Both `cin.getline()` and `cin.get()` can read the input of one whole line until reaching a newline character `'\n'`.

```
// cin.getline() reads up to (n-1) characters into str and adds a null character at the end; if it  
//encounters the delimiter (i.e., '\n'), it will also read '\n' and then automatically replace it with null.  
istream& getline(char* str, streamsize n, char delim = '\n');
```

```
// Reads a character and stores it in the provided reference
```

```
istream& get(char& ch);
```

```
// Reads up to (n-1) characters into str, stopping at delim or max length (n-1)
```

```
// But it will not read '\n', and leave it in the input queue
```

```
istream& get(char* str, streamsize n, char delim = '\n');
```

Function
Overloading

Input/Output – Handle String

- `cin.getline()`

```
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.getline(name, ArSize); // reads through newline
    cout << "Enter your favorite dessert:\n";
    cin.getline(dessert, ArSize);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Code:

```
char name[10];
cout << "Enter your name: ";
cin.getline(name, 10);
```

User responds by typing **Jud**, then pressing **ENTER**

Enter your name: Jud **ENTER**

`cin.getline()` responds by reading Jud, reading the newline generated by the Enter key, and replacing it with a null character.



newline replaced with a null character.

Sample Output

Enter your name:

Dirk Hammernose **Enter**



Enter your favorite dessert:

Radish Torte **Enter**

I have some delicious Radish Torte for you, Dirk Hammernose.

Input/Output – Handle String

- `cin.get()`

```
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.get(name, ArSize); // read the string
    cin.get(); // read the newline character '\n' at the end
    cout << "Enter your favorite dessert:\n";
    cin.get(dessert, ArSize).get(); // function concatenate, as cin.get(dessert,
    ArSize) will return the object cin
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";

    return 0;
}
```

Sample Output

```
Enter your name:
Yong Wang 
Enter your favorite dessert:
cake 
I have some delicious cake for you, Yong Wang.
```



Data Types

- **Basic Types:** C++ and C share the same basic data types, e.g., char, int, short, long, float, double, bool, etc.

```
// limits.cpp -- some integer limits
#include <iostream>
#include <climits> // max/min value symbols are defined in climits file
int main()
{
    using namespace std;
    short n_short = SHRT_MAX;
    int n_int = INT_MAX; // initialize n_int to max int value symbols defined in climits file
    long n_long = LONG_MAX;
    long long n_llong = LLONG_MAX;

    // sizeof operator yields size
    cout << "int is " << sizeof(int) << " bytes." << endl;
    cout << "short is " << sizeof(short) << " bytes." << endl;
    cout << "long is " << sizeof(long) << " bytes." << endl;
    cout << "long long is " << sizeof(long long) << " bytes." << endl;
    cout << endl;

    cout << "Maximum values:" << endl;
    cout << "int: " << n_int << endl;
    cout << "short: " << n_short << endl;

    cout << "long: " << n_long << endl;
    cout << "long long: " << n_llong << endl << endl;
    cout << "Minimum int value = " << INT_MIN << endl;
    cout << "Bits per byte = " << CHAR_BIT << endl;

    return 0;
}
```

Output

```
int is 4 bytes.
short is 2 bytes.
long is 8 bytes.
long long is 8 bytes.
```

```
Maximum values:
int: 2147483647
short: 32767
long: 9223372036854775807
long long: 9223372036854775807
```

```
Minimum int value = -2147483648
Bits per byte = 8
```

- **Compound Types:** C++ and C are similar in the majority of compound types, e.g., arrays, structures, unions, strings, etc.
- But there are also some differences, for example,
 - structures
 - unions
 - string-class strings
 - reference
 - ...

Structure in C program

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

// Define a structure
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    struct Person p1 = {"Alice", 25, 175};
    printf("Name: %s, Age: %d, Height: %.1f\n", p1.name, p1.age, p1.height);

    return 0;
}
```

Structure in C++ program

```
#include <iostream>
using namespace std;

struct Person {
    char name[50];
    int age;
    float height;

    //Difference 1: A structure in C++ can have member functions --- OOP
    void display() {
        cout << "Name: " << name << ", Age: " << age << ", Height: " << height << endl;
    }
};

int main() {
    Person p1 = {"Alice", 25, 175}; //Difference 2: No need to use `struct` keyword here in C++

    p1.display();
    return 0;
}
```

Union

- A union is a data format that can hold **different data types** but **only one type at a time**. All these members **share the same memory address**.
- Unions are memory-efficient, and use only **the largest member's size**.

```
#include <iostream>
using namespace std;
```

```
union Data {
    short sValue;
    double dValue;
```

```
void printShort() { cout << "sValue: " << sValue << endl; }
void printDouble() { cout << "dValue: " << dValue << endl; }
```

```
};
```

```
int main() {
    Data data;
```

```
cout << "Size of Data: " << sizeof(Data) << " bytes."<<endl;
cout << "Size of Short: " << sizeof(short) <<" bytes." <<endl;
cout << "Size of Double: " << sizeof(double) << " bytes." <<endl<<endl;
```

```
data.sValue = 42;
cout << "Short: " << data.sValue << endl;
data.printShort();
```

```
data.dValue = 3.14; // Overwrites `sValue`
cout << "Double: " << data.dValue << endl;
data.printDouble();
```

```
return 0;
```

```
}
```

Output

Size of Data: 8 bytes.
Size of Short: 2 bytes.
Size of Double: 8 bytes.

Short: 42
sValue: 42
Double: 3.14
dValue: 3.14

Key differences from unions in C:

- Unions in C++ can have member functions (object-oriented programming)
- The keyword "union" is not needed when declaring a union object

- When will we use unions?
 - Unions often (but not exclusively) are used to **save memory space**
 - Example application scenarios:
 - Embedding systems where space is a critical consideration: the processors used to control a toaster oven, an MP3 player, or a Mars rover.
 - Operating systems
 - Hardware data structures

String

- C++ has two ways to deal with strings:
 - *C-style string*, which is taken from C, i.e., a **character array ending with a null character `\0`**.
 - *string class* from the standard library in C++

String Class

#include <string>

Function	Task
append()	Appends a part of string to another string.
at()	Obtains the character stored at a specified location.
compare()	Compares string against the invoking string.
empty()	Returns true if the string is empty; otherwise returns false.
erase()	Removes character as specified.
find()	Searches for the occurrence of a specified substring.
insert()	Inserts character at specified location.
length()	Gives number of elements in a string.
replace()	Replace specified characters with a given string.

More functions of string class of C++ can be found here:

<https://cplusplus.com/reference/string/string/?kw=string>

String Class

Output:

s3: 12345abcde

12345

12345

After insert: 1234abcde5

After erase: 12345

After replace: a12345e

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s1("12345");
    string s2("abcde");

    // Concatenation
    string s3 = s1 + s2;
    cout<< "s3: "<<s3<<endl;

    // Printing each character of s1 using .at() and []
    for (int i = 0; i < s1.length(); i++)
        cout << s1.at(i);
    cout << endl;

    for (int j = 0; j < s1.length(); j++)
        cout << s1[j];
    cout << endl;

    // Insert a string s2 into s1 at position 4
    s1.insert(4, s2); // s1 becomes "1234abcde5"
    cout << "After insert: " << s1 << endl;

    // Remove 5 characters from s1 starting from position 4
    s1.erase(4, 5); // s1 becomes "12345"
    cout << "After erase: " << s1 << endl;

    // Replace 3 characters in s2 with s1 starting from position 1
    s2.replace(1, 3, s1); // s2 becomes "a12345e"
    cout << "After replace: " << s2 << endl;

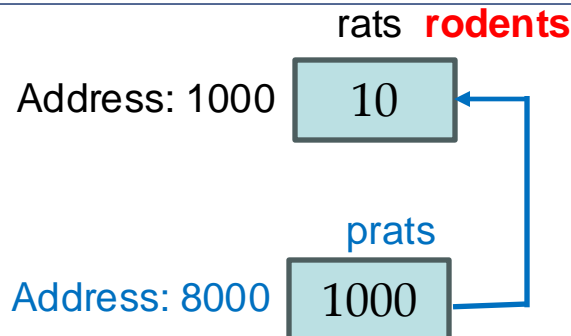
    return 0;
}
```

Reference Variable

- The reference variable is a new compound type added by C++
- A reference is essentially **an alias**, or an alternative name, for a previously defined variable. The reference and original variable name can be used interchangeably
- You should initialize a reference variable when you declare it. After that, you cannot change the reference to refer to another variable

```
int rat;
int & rodent;
rodent = rat; // No, you can't do this.
```

```
int rats = 10;
int & rodents = rats; // rodents: a reference
int * prats = &rats; // prats: a pointer
```



Reference vs. Pointer

```
int & rodents = rats;
is, in essence, equivalent to
int * const rodents = &rats
```

Reference Variable

```
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats; // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    cout << "rats address = " << &rats;
    cout << ", rodents address = " << &rodents << endl;

    int bunnies = 50;
    rodents = bunnies; // can we change the reference?
    cout << "bunnies = " << bunnies;
    cout << ", rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    cout << "bunnies address = " << &bunnies;
    cout << ", rodents address = " << &rodents << endl;
    return 0;
}
```

Output

```
rats = 101, rodents = 101
rats address = 0x7fff42eab6c4, rodents address = 0x7fff42eab6c4
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0x7fff42eab6c0, rodents address = 0x7fff42eab6c4
```

Recall: Constant Pointer vs. Pointer to Constant

- Take `int` pointer as an example (Y—Changeable, N—Non-changeable)

Description	Declaration	*ptr	ptr
pointer pointing to a const entity	<code>const int * ptr</code>	N	Y
const pointer	<code>int * const ptr</code>	Y	N
const pointer pointing to a const entity	<code>const int * const ptr</code>	N	N

```
#include <iostream>

int main() {
    int value1 = 10;
    int value2 = 20;

    int* const ptr1 = &value1;
    *ptr1 = 15; // Allowed
    ptr1 = &value2; // Not allowed

    const int* ptr2 = &value1;
    ptr2 = 15; // Not allowed
    ptr2 = &value2;

    const int* const ptr3 = &value1;
    *ptr3 = 15; // Not allowed
    ptr3 = &value2; // Not allowed

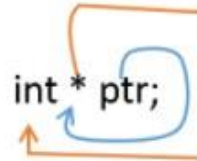
    return 0;
}
```

Recall: Constant Pointer vs. Pointer to Constant

For those who don't know about Clockwise/Spiral Rule: Start from the name of the variable, move clockwise (in this case, move backward) to the next **pointer** or **type**. Repeat until expression ends.

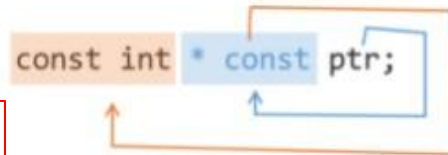
- Clockwise/Spiral Rule

Here is a demo:



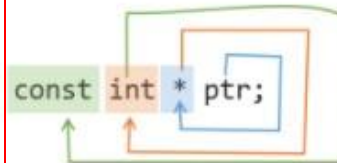
```
int * ptr;
```

ptr is a **pointer** to **int**



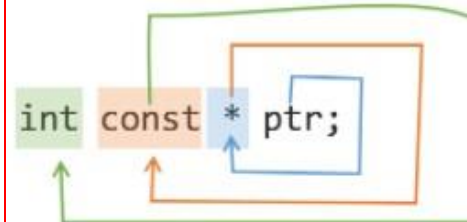
```
const int * const ptr;
```

ptr is a **constant pointer** to **const int**



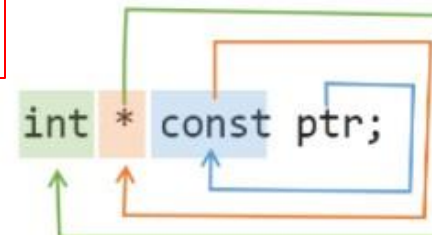
```
const int * ptr;
```

ptr is a **pointer** to **int constant** (i.e. **const int**)



```
int const * ptr;
```

ptr is a **pointer** to **const int**



```
int * const ptr;
```

ptr is a **const pointer** to **int**

If the word **const** appears to the left of the asterisk, what's pointed to is constant;

If the word **const** appears to the right of the asterisk, the pointer itself is constant;

If **const** appears on both sides, both are constant.

References as Function Parameters

- References are used as function parameters, making the variable name in a function an alias for a variable in the calling program. Thus, **the changes done via the references within the function** will also **be reflected outside the function**
- Such method of passing arguments is called **passing by reference**

```
#include <iostream>
void swapr(int &a, int &b);    // a, b are aliases for ints
void swapp(int *p, int *q);  // p, q are addresses of ints
void swapv(int a, int b);    // a, b are new variables
int main()
{
    using namespace std;
    int wallet1 = 300;
    int wallet2 = 350;

    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using references to swap contents:\n";
    swapr(wallet1, wallet2);    // pass variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using pointers to swap contents again:\n";
    swapp(&wallet1, &wallet2); // pass addresses of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Trying to use passing by value:\n";
    swapv(wallet1, wallet2);    // pass values of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;
    return 0;
}
```

```
void swapr(int &a, int &b)    // use references
{
    int temp;

    temp = a;                // use a, b for values of variables
    a = b;
    b = temp;
}
```

Successful swapping

References as Function Parameters

- References are used as function parameters, making the variable name in a function an alias for a variable in the calling program. Thus, **the changes done via the references within the function** will also **be reflected outside the function**
- Such method of passing arguments is called **passing by reference**

```
#include <iostream>
void swapr(int &a, int &b);    // a, b are aliases for ints
void swapp(int *p, int *q);  // p, q are addresses of ints
void swapv(int a, int b);    // a, b are new variables
int main()
{
    using namespace std;
    int wallet1 = 300;
    int wallet2 = 350;

    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using references to swap contents:\n";
    swapr(wallet1, wallet2);    // pass variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using pointers to swap contents again:\n";
    swapp(&wallet1, &wallet2); // pass addresses of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Trying to use passing by value:\n";
    swapv(wallet1, wallet2);    // pass values of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;
    return 0;
}
```

```
void swapp(int *p, int *q)    // use pointers
{
    int temp;
    temp = *p;                // use *p, *q for values of variables
    *p = *q;
    *q = temp;
}

void swapv(int a, int b)      // try using values
{
    int temp;
    temp = a;                 // use a, b for values of variables
    a = b;
    b = temp;
}
```

Successful swapping

Failed swapping

When to Use Reference Arguments?

- There are two main reasons for using reference arguments:
 - To allow you to **alter a data object** in the calling function
 - To **speed up a program** by passing a reference instead of an entire data object, especially when the data object is large
- To avoid unexpected data change, you should **declare reference arguments as `const` whenever it is appropriate to do so**

Const Reference

```
const int & rodents = rats;
```

is, in essence, equivalent to

```
const int * const rodents = &rats
```

When to Use Reference Arguments?

- To avoid unexpected data change, you should **declare reference arguments as const** whenever it is appropriate to do so

```
#include <iostream>

// passing by reference
double calVolume(double& a) {
    a = a * a * a;
    return a;
}

int main() {
    double x = 3.0;
    double volume = calVolume(x);

    std::cout << "Original side length: " << x << std::endl; // Outputs: 27
    std::cout << "Volume of the cube: " << volume << std::endl; // Outputs: 27

    return 0;
}
```

```
#include <iostream>

// passing by reference
// Use const reference to avoid unexpected variable change
double calVolume(const double& a) {
    double volume = a * a * a;
    return volume;
}
```

Dynamic Memory

- Dynamic memory refers to the **memory that is allocated during runtime**
- There are cases when the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input
- In C++, we use the **new** operator for allocating dynamic memory, and the **delete** operator for freeing previously-allocated dynamic memory

```
// Allocate memory for a variable  
typeName pointerName = new typeName;  
  
// Allocate memory for a dynamic array  
typeName pointerName = new typeName [ElementNum]
```

```
// Free the memory of a dynamically-created variable  
delete pointerName;  
  
// Free the memory of a dynamic array  
delete []pointerName;
```

Dynamic Memory

```
#include <iostream>
using namespace std;

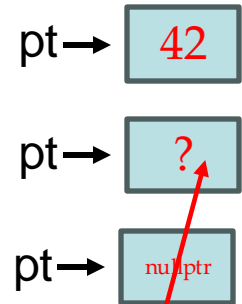
int main() {
    int* pt = new int; // Allocate memory for an integer

    *pt = 42;
    cout << "Value: " << *pt << ", memory address: " << pt << endl;

    delete pt; // Free memory
    cout << "Value: " << *pt << ", memory address: " << pt << endl;

    pt = nullptr; // prevents dangling pointer, nullptr: null pointer
    cout << "memory address: " << pt << endl;

    return 0;
}
```



Output

```
Value: 42, memory address: 0x1a9ea2b0
Value: 109034, memory address: 0x1a9ea2b0
memory address: 0
```

It is actually a random value

Dynamic Memory

```
#include <iostream>
using namespace std;

int main() {
    int size;
    cout << "Enter the number of elements: ";
    cin >> size;

    // Allocate memory for an array dynamically
    int* arr = new int[size];

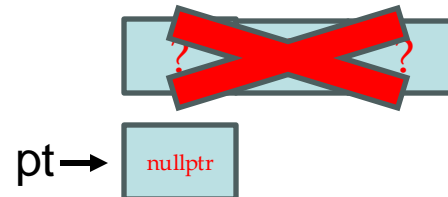
    // Input values
    cout << "Enter " << size << " elements: ";
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }

    // Display values
    cout << "You entered: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Deallocate memory
    delete[] arr;
    arr = nullptr; // Prevent dangling pointer

    return 0;
}
```

Size = 3



Output

```
Enter the number of elements: 3

Enter 3 elements: 10 20 30

You entered: 10 20 30
```

Dynamic Memory

- Do remember to use `new` and `delete` in parallel to **avoid memory leakage**
 - Free the dynamic memory when your program does not need it
- Avoid common mistakes or risky operations

```
int * pt = new int; //ok
delete pt; //ok
delete pt; //not ok
```

```
int jugs = 5; //ok
int *ps = &jugs; //ok
delete ps; //not ok
```

```
int *pt = new int;
short *ps = new short [500];
```

```
// undefined effect, don't do it
delete [] pt;
```

```
// undefined effect, don't do it
delete ps;
```

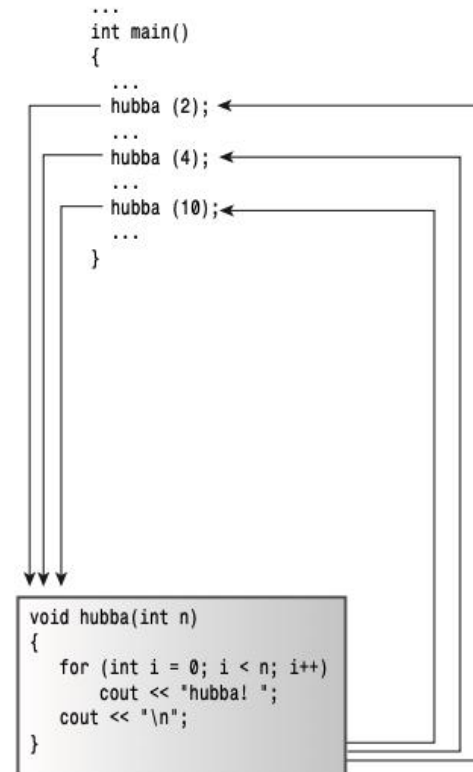
- Input/output
- Data types:
 - Basic data types
 - Compound data types
- Reference variables
- Dynamic memory allocation
- Functions
 - Inline functions
 - Default arguments
 - Function overloading
 - Function templates

Inline Function

- **Inline functions** are a C++ enhancement designed to speed up programs.
- There are overheads in calling regular functions, as the program need to
 - store memory address of subsequent instructions
 - copy function arguments to the stack
 - jump to memory location of the function
 - ...
- For inline functions, the compiler **replaces function call with the corresponding function code** and there is no need to jump back and forth

Inline Function

- **Inline functions** are often for **simple tasks with short code execution time**, so that the inline call can save a large portion of the time used by the non-inline call
- The compiler **may process an inline function as a regular function** when it is too large



A regular function transfers program execution to a separate function.

```
...
int main()
{
    ...
    {
        n = 2;
        for (int i = 0; i < n; i++)
            cout << "hubba! ";
        cout << "\n";
    }
    ...
    {
        n = 4;
        for (int i = 0; i < n; i++)
            cout << "hubba! ";
        cout << "\n";
    }
    ...
    {
        n = 10;
        for (int i = 0; i < n; i++)
            cout << "hubba! ";
        cout << "\n";
    }
    ...
}
```

An inline function replaces a function call with inline code.

Inline Function

```
#include <iostream>
using namespace std;

// an inline function definition
inline double square(double x) { return x * x; }

int main() {

    double a, b;
    double c = 13.0;
    a = square(5.0);
    b = square(4.5 + 7.5);
    cout << "a = " << a << ", b = " << b << endl;
    cout << "c = " << c;
    cout << ", c squared = " << square(c++) << endl;
    cout << "Now c = " << c << endl;
    return 0;
}
```

Output

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

Default Arguments

- A **default argument** is a value that's used automatically, if you omit the corresponding actual argument from a function call.

```
#include<iostream>
using namespace std;

int add(int x = 5,int y = 6) {
    return x + y;
}

int main() {
    cout<< add(10,20)<<endl; //10+20, 30
    cout<<add(10) <<endl;    //10+6, 16
    cout<<add() <<endl;      //5+6, 11

    return 0;
}
```

Default Arguments

- When you declare/define a function with an argument list, you must **add defaults from right to left**. That is, you cannot provide a default value for a particular argument unless you also provide defaults for all the arguments to its right:

```
int add(int x, int y = 5, int z = 6); //valid
int add(int x = 1, int y = 5, int z); //invalid
int add(int x = 1, int y, int z = 6); //invalid
```

- When using the a function with default arguments, the actual arguments are assigned to the corresponding formal arguments **from left to right; you can't skip over arguments**.

```
int add(int x, int y = 5, int z = 6);

//For the above function prototype
sum = add(2);           //same as add(2,5,6)
sum = add(1,8);         //same as add(1,8,6)
sum = add(8,7,6);       //no default arguments used
sum = add(8, ,6);       //invalid, doesn't set y to 5
```

Default Arguments

- Default arguments must be specified in **function declarations, and avoid repeating default arguments in the function's definition** to prevent redefinition errors
- It is also fine to remove function declaration/prototype, and **put the function definition with default arguments before the main function**

```
//function declaration with
default arguments
int add(int x = 5,int y = 6);

int main() {
    add();
}

//No default arguments in the
function definition
int add(int x,int y) {
    return x + y;
}
```

```
// Only function definition
int add(int x = 5,int y = 6) {
    return x + y;
}

int main() {
    add();
}
```

Function Overloading

- C++ allows multiple functions to share the same name, which is often called **function overloading** or **function polymorphism**
- We can use function overloading to design multiple functions that do the same thing, but with **different argument lists (a.k.a, function signature)**. Different argument lists refer to different **argument types** or **argument numbers**.

```
int add(int x, int y);  
float add(float x, float y);
```

Different argument types

```
int add(int x, int y);  
int add(int x, int y, int z);
```

Different argument numbers

- The function-matching process does discriminate between `const` and `non-const` variables.

```
void dribble(char * bits);  
void dribble (const char *cbits);
```

Function Overloading

- Below are some examples of **invalid** function overloading

```
int add(int x, int y);  
int add(int a, int b);
```

Argument names do not matter

```
int add(int x, int y);  
void add(int a, int b);
```

The return value does not matter

```
double cube(double x);  
double cube(double & x);
```

The compiler cannot know which function prototype to use when function call "cube(x); " is seen.

Function Templates

- A *function template* is a generic function description. It defines a function in terms of a generic type for which a specific type, such as `int` or `double`, can be substituted.
- With function templates, we can easily implement the same function for **multiple data types**.

```
template <class Any>
void Swap(Any &a, Any &b) {
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
template <typename T>
void Swap(T &a, T &b) {
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

The keyword “**class**” can be replaced by “**typename**”.

Function Templates

```
#include <iostream>

template <typename T>
T getMax(T a, T b) {
    return (a < b) ? b : a;
}

int main() {
    int i = 5, j = 10;
    std::cout << "Max(i, j): " << getMax(i, j) << std::endl;

    double x = 5.5, y = 2.3;
    std::cout << "Max(x, y): " << getMax(x, y) << std::endl;

    char c1 = 'a', c2 = 'z';
    std::cout << "Max(c1, c2): " << getMax(c1, c2) << std::endl;

    return 0;
}
```

Output

```
30
16
11
```

The image shows the front cover of the book 'C++ Primer Plus, Fifth Edition' by Stephen Prata. The cover is a solid blue color. At the top, the text 'C++' is written in a large, white, sans-serif font. Below it, 'Primer Plus' is written in a slightly smaller, white, serif font. Underneath that, 'Fifth Edition' is written in a small, white, sans-serif font. In the bottom left corner, the 'SAMS' logo is visible in white. In the bottom right corner, the author's name 'Stephen Prata' is written in white.

C++ Primer Plus Fifth Edition

SAMS

Stephen Prata

References:

- [1] Prata, Stephen. C++ primer plus.
Sams Publishing, 2002, 5th edition.
Chapters 1, 2, 3, 4, 8.

Questions?

Thank You !