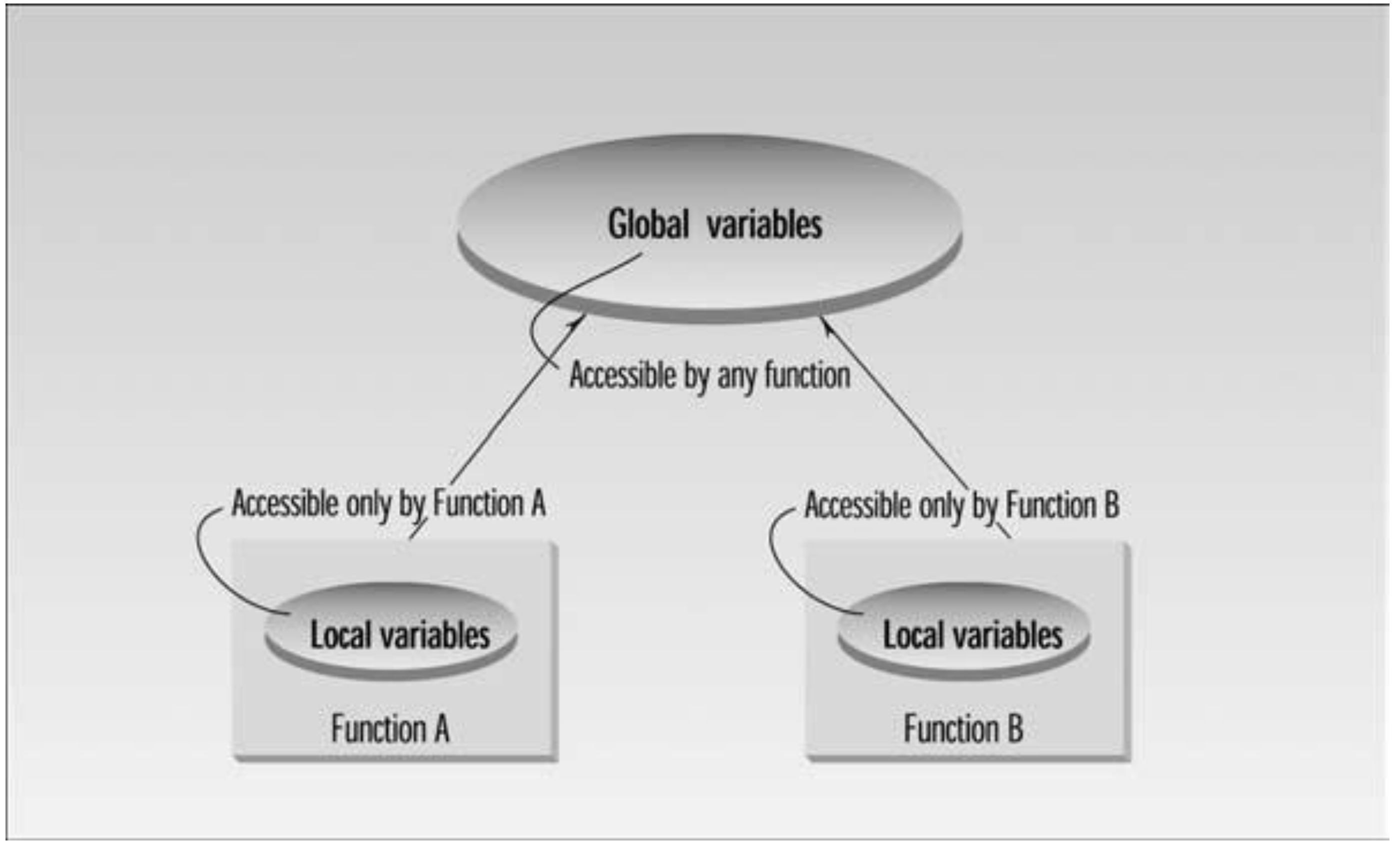
# Object Oriented Programming with C++

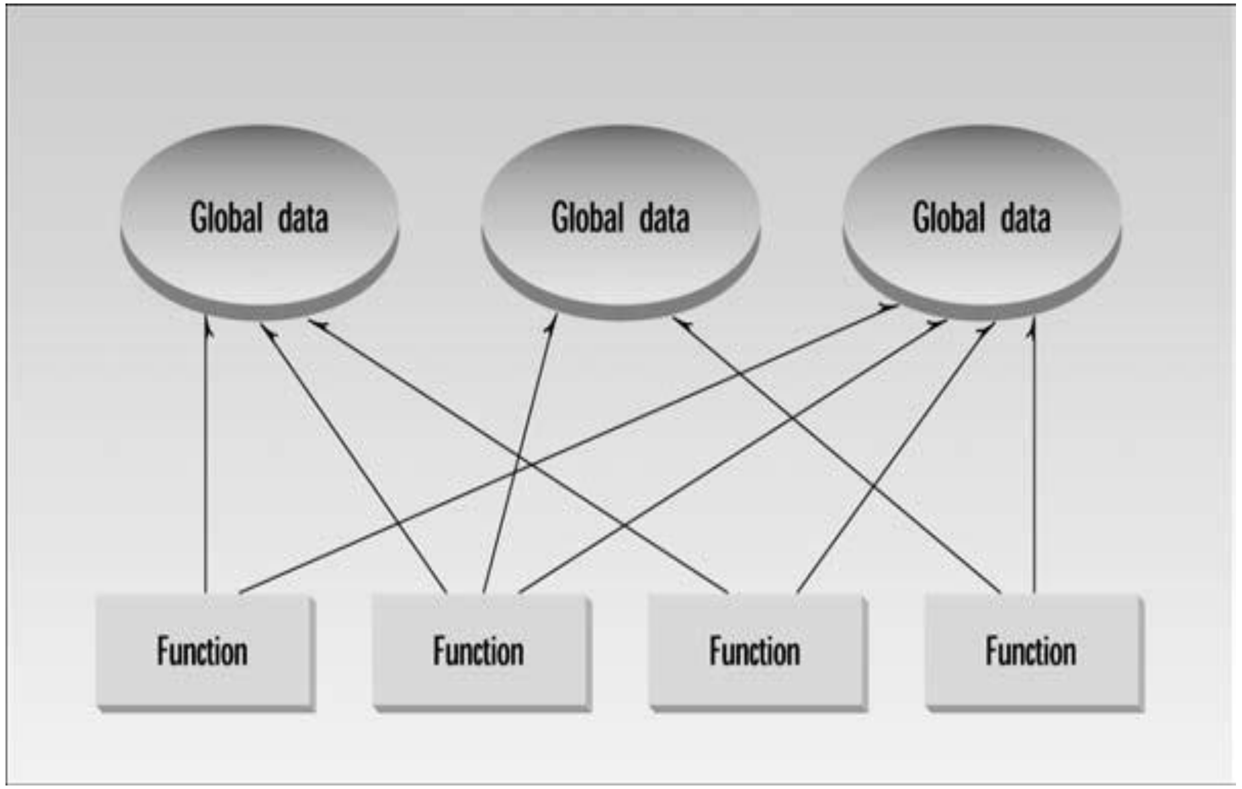
## Problems with Procedure Languages

- **Unrestricted Access**


Global variables
Accessible by any function
Accessible only by Function A
Accessible only by Function B
Local variables
Local variables
Function A
Function B

## Problems with Procedure Languages

- **Unrestricted Access**
  - First, it makes a program's structure difficult to conceptualize.
  - Second, it makes the program difficult to modify. A change made in a global data item may necessitate rewriting all the functions that access that item.



For example, in our inventory program, someone may decide that the product codes for the inventory items should be changed from 5 digits to 12 digits. This may necessitate a change from a short to a long data type.

## Problems with Procedure Languages

- **Real-World Modeling**
- The arrangement of separate data and functions does a poor job of modeling things in the real world.
- In the physical world we deal with objects which are not like functions as they have both *attributes* and *behavior*.

- **Attributes**
  - Referred to as characteristics.
  - Equivalent to data in a program as they have a certain specific values.
  - Examples for people, eye color and job title; and, for cars, horsepower and number of doors.

- **Behavior**
  - Behavior is something a real-world object does in response to some stimulus.
  - Behavior is like a function: you call a function to do something (display the inventory, for example) and it does it.
  - If you apply the brakes in a car, it will generally stop.

## The Object-Oriented Approach

- The fundamental idea behind object-oriented languages is to combine into a single unit both *data* and the *functions that operate on that data*.
- Such a unit is called an *object*.
- An object's functions, called *member functions* in C++, typically provide the only way to access its data.
- If you want to read a data item in an object, you call a member function in the object.
- It will access the data and return the value to you.
- You can't access the data directly.
- The data is *hidden*, so it is safe from accidental alteration. Data and its functions are said to be *encapsulated* into a single entity.

## The Object-Oriented Approach

- If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object.
- No other functions can access the data.
- This simplifies writing, debugging, and maintaining the program.

## Basic Code

```
#include<stdio.h>


main()
{
        printf("Hello World");
}
```

```
#include<iostream>


main()
{
        std::cout<<"Hello World";
}
```

## Code

- main() function
- whitespaces
- cout
- <<
- preprocessor
- preprocessor directive
- namespace
- comments
- variables

**Reading Assignment: Data types**

## Identifiers

- A valid identifier is a sequence of one or more letters, digits, or underscore characters (_).
- Spaces, punctuation marks, and symbols cannot be part of an identifier.
- In addition, identifiers shall always begin with a letter.
  - They can also begin with an underline character (_), but such identifiers are -on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere.
  - In no case can they begin with a digit.
- C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords.

## Output/Error

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

main()
{
int main=20;
cout<<main;
}
```

```
OUTPUT
20
```

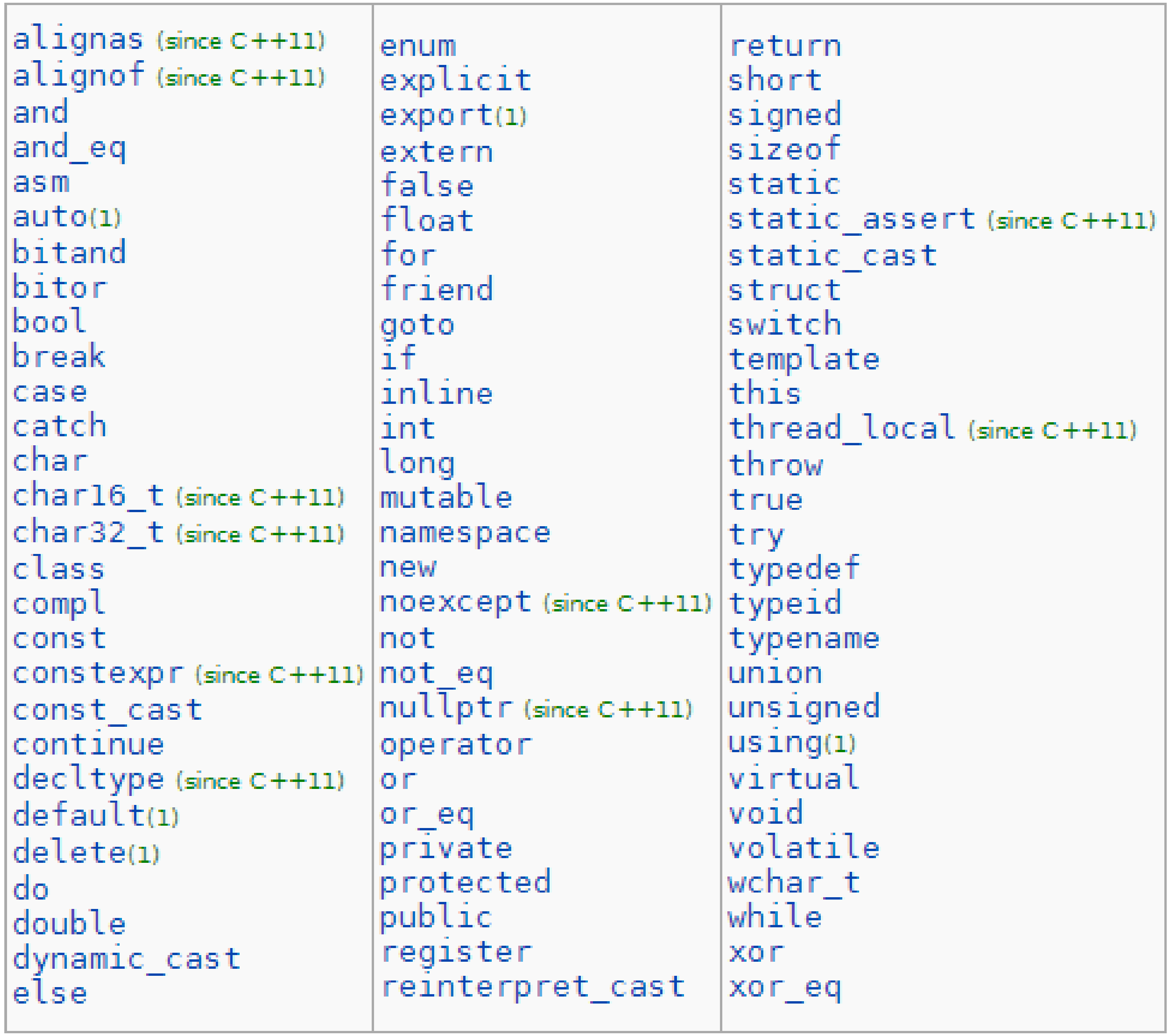
## Output/Error

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

void main()
{
int x=20;
cout<<x;
}
```

Error

In older versions of C++ you could give main() the return type of void and dispense with the return statement, but this is not considered correct in Standard C++.

## List of Keywords



| | | |
|---|---|---|
| alignas (since C++11) | enum | return |
| alignof (since C++11) | explicit | short |
| and | export(1) | signed |
| and_eq | extern | sizeof |
| asm | false | static |
| auto(1) | float | static_assert (since C++11) |
| bitand | for | static_cast |
| bitor | friend | struct |
| bool | goto | switch |
| break | if | template |
| case | inline | this |
| catch | int | thread_local (since C++11) |
| char | long | throw |
| char16_t (since C++11) | mutable | true |
| char32_t (since C++11) | namespace | try |
| class | new | typedef |
| compl | noexcept (since C++11) | typeid |
| const | not | typename |
| constexpr (since C++11) | not_eq | union |
| const_cast | nullptr (since C++11) | unsigned |
| continue | operator | using(1) |
| decltype (since C++11) | or | virtual |
| default(1) | or_eq | void |
| delete(1) | private | volatile |
| do | protected | wchar_t |
| double | public | while |
| dynamic_cast | register | xor |
| else | reinterpret_cast | xor_eq |

## Output/Error

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

main()
{
int BREAK=20;
cout<<BREAK;
}
```

```
OUTPUT

20
```

## Output/Error

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

int main (){
  int a=5;
  int b(3);
  int c{2};
  int result;

  a = a + b;
  result = a - c;
  cout << result;
  return 0;
}
```

OUTPUT

6

## Initialization of variables

- The first one, known as c-like initialization (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:

  type identifier = initial_value;

- A second method, known as constructor initialization (introduced by the C++ language), encloses the initial value between parentheses (()):

  type identifier (initial_value);

- A third method, known as uniform initialization, similar to the above, but using curly braces ({}) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011)

  type identifier {initial_value};

## Output/Error

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

main()
{
int a=0;
auto b=a;
cout<<a<<endl<<b<<endl;
}
```

OUTPUT

```
0
0
```

## Output/Error

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

main()
{
int a=0;
decltype (a)b;
cout<<a<<endl<<b<<endl;
}
```

OUTPUT

```
0
Garbage
```

## Type deduction: auto and decltype

- When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use auto as the type specifier for the variable:

```
int foo = 0;
auto bar = foo;  // the same as: int bar = foo;
```

- Here, bar is declared as having an auto type; therefore, the type of bar is the type of the value used to initialize it: in this case it uses the type of foo, which is int.

- Variables that are not initialized can also make use of type deduction with the decltype specifier:

```
int foo = 0;
decltype(foo) bar;  // the same as: int bar;
```

- Here, bar is declared as having the same type as foo.

## Constants

- const double pi = 3.1415926;
- #define PI 3.14159

## Output/Error

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

main()
{
int a;
cout<<(a=7);
}
```

```
OUTPUT

7
```

## Output/Error

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

main()
{
int b;
int a=2+(b=5);

cout<<a<<endl<<b<<endl;
}
```

OUTPUT

```
7
5
```

## Output/Error

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

main()
{
float a, b, c, d;
a=5/2;
b=5/2.0;
c=5.0/2;
d=5.0/2.0;
cout<<a<<endl<<b<<endl<<c<<endl<<d;
}
```

OUTPUT

```
2
2.5
2.5
2.5
```

## Output/Error

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

main()
{
float a,b,c,d;
a=5%2;
b=5%2.0;
c=5.0%2;
d=5.0%2.0;
cout<<a<<endl<<b<<endl<<c<<endl<<d;
}
```

ERROR

Invalid operands of types 'double' and 'double' to binary operator %

## Output/Error

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

main(){
int a,b,c,d;
a=5/2;
b=5/-2;
c=-5/2;
d=-5/-2;
cout<<a<<endl<<b<<endl<<c<<endl<<d;
}
```

OUTPUT

```
2
-2
-2
2
```

## Output/Error

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

main()
{
int a,b,c,d;
a=5%2;
b=5%-2;
c=-5%2;
d=-5%-2;
cout<<a<<endl<<b<<endl<<c<<endl<<d;
}
```

OUTPUT

```
1
1
-1
-1
```

## Output/Error

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

main()
{
int a=1,2,3,4,5;
cout<<a;
}
```

Error

expected unqualified-id before numeric constant

## Output/Error

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

main()
{
int a;
a=1,2,3,4,5;
cout<<a;
}
```

OUTPUT

```
1
```

## Output/Error

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

main()
{
int a=(1,2,3,4,5);
cout<<a;
}
```

OUTPUT

5

## Output/Error

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

main()
{
int a=6;
if(a&1)
cout<<"ODD";
else
cout<<"EVEN";
}
```

OUTPUT

ODD

## Operators

- Assignment operator (=)
- Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
- Relational and comparison operators ( ==, !=, >, <, >=, <= )
- Logical operators ( !, &&, || )
- Conditional ternary operator ( ?: )
- Comma operator ( , )
- Bitwise operators ( &, |, ^, ~, <<, >> )
- sizeof

| Level | Precedence group | Operator | Description | Grouping |
|---|---|---|---|---|
| 1 | Scope | :: | scope qualifier | Left-to-right |
| 2 | Postfix (unary) | ++ -- | postfix increment / decrement | Left-to-right |
| | | () | functional forms | |
| | | [] | subscript | |
| | | . -> | member access | |
| 3 | Prefix (unary) | ++ -- | prefix increment / decrement | Right-to-left |
| | | ~ ! | bitwise NOT / logical NOT | |
| | | + - | unary prefix | |
| | | & * | reference / dereference | |
| | | new delete | allocation / deallocation | |
| | | sizeof | parameter pack | |
| | | (type) | C-style type-casting | |
| 4 | Pointer-to-member | .* ->* | access pointer | Left-to-right |
| 5 | Arithmetic: scaling | * / % | multiply, divide, modulo | Left-to-right |
| 6 | Arithmetic: addition | + - | addition, subtraction | Left-to-right |
| 7 | Bitwise shift | << >> | shift left, shift right | Left-to-right |

| Level | Precedence group | Operator | Description | Grouping |
|---|---|---|---|---|
| 8 | Relational | < > <= >= | comparison operators | Left-to-right |
| 9 | Equality | == != | equality / inequality | Left-to-right |
| 10 | And | & | bitwise AND | Left-to-right |
| 11 | Exclusive or | ^ | bitwise XOR | Left-to-right |
| 12 | Inclusive or | \| | bitwise OR | Left-to-right |
| 13 | Conjunction | && | logical AND | Left-to-right |
| 14 | Disjunction | \|\| | logical OR | Left-to-right |
| 15 | Assignment-level expressions | = *= /= %= += -= >>= <<= &= ^= \|= | assignment / compound assignment | Right-to-left |
|  |  | ?: | conditional operator |  |
| 16 | Sequencing | , | comma separator | Left-to-right |

## Code

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

main()
{
int a=6;
cout<<"a="<<"\n"<<a;
}
```

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

main()
{
int a=6;
cout<<"a="<<endl<<a;
}
```

The endl manipulator produces a newline character, exactly as the insertion of '\n' does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already.

## Reading Input

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

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".
\n";
  return 0;
}
```

- Please enter an integer value: 12
  - The value you entered is 12 and its double is 24.
- Please enter an integer value: e
  - The value you entered is 0 and its double is 0.
- Please enter an integer value: str
  - The value you entered is 0 and its double is 0.

What happens in the example above if the user enters something else that cannot be interpreted as an integer? Well, in this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable i, producing undetermined results if the value of i is used later.

## Code

```
#include<stdio.h>

main()
{
int x;
scanf("%d",&x);
if(x%2==0)
printf("Even");
else
printf("Odd");
}
```

```
#include<stdio.h>
void even_odd(int);

main(){
int x;
scanf("%d",&x);
even_odd(x);
}

void even_odd(int y){
if(y%2==0)
printf("Even");
else
printf("Odd");
}
```

## C++ Code

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

class test{
int x;
public:
void get(){
cin>>x;
}
void even_odd(){
if(x%2==0)
cout<<"Even";
else
cout<<"Odd";
}}ob;
```

```
main()
{
//test ob;
ob.get();
ob.even_odd();
}
```

## Structure of a Class

```
class class_name {
 access_specifier_1:
   member1;
   member2;
 access_specifier_2:
   member1;
   member2;   ...
} object_names;
```

## Code

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

class odd_even{
int x;
public:
        void get(){
                x=10;
        }
        void put(){
                cout<<"x="<<x;
        }
};
```

```
int main()
{
odd_even ob;
ob.get();
ob.put();
}
```

OUTPUT

```
X=10
```

## Code

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

class odd_even{
int x;
public:
void get();
void put();
};

void odd_even:: get(){
x=10;
}

void odd_even::  put(){
cout<<"x="<<x;
}

int main()
{
odd_even ob;
ob.get();
ob.put();
}
```

OUTPUT

```
X=10
```

## Code

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

class odd_even
{
int x;
public:
void get();
void put();
};

void odd_even:: get(){
x=10;
}
```

```
void odd_even:: put(){
cout<<"x="<<x;
}


int main()
{
odd_even ob;
ob.get();
ob.put();
cout<<ob.x;
}
```

Error

```
test.cpp: In function 'int main()':
test.cpp:25:10: error: 'int
odd_even::x' is private within this
context
 cout<<ob.x;
          ^
test.cpp:6:5: note: declared private
here
 int x;
     ^
```

## Code

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

class odd_even{
int x;
public:
void get();
void put();
};

void odd_even:: get(){
x=10;
put();
}
```

```
void odd_even::  put(){
cout<<"x="<<x;
}


int main()
{
odd_even ob;
ob.get();
}
```

OUTPUT

```
X=10
```

## Code

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

class odd_even{
int x;
public:
void get();
void put();
};

inline void odd_even:: get(){
x=10;
}
```

```
inline void odd_even::  put(){
cout<<"x="<<x;
}


int main()
{
odd_even ob;
ob.get();
ob.put();
}
```

OUTPUT

```
X=10
```

## Code

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

class odd_even{
int x;
void get();
void put();
};

inline void odd_even:: get(){
x=10;
}
```

```
inline void odd_even::  put(){
cout<<"x="<<x;
}


int main()
{
odd_even ob;
ob.get();
ob.put();
}
```

Error

```
test.cpp: In function 'int main()':
test.cpp:23:8: error: 'void
odd_even::get()' is private within
this context
ob.get();
       ^
test.cpp:11:13: note: declared
private here
inline void odd_even:: get(){
```

```
#include<iostream>
using namespace std;
int main()
{
int x;
float y;

cout<<"x="<<x<<"\ny="<<y;
}
```

OUTPUT

```
x=4201003
y=8.15556e-043
```

## Code

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

class odd_even{
int x[10];
int s_odd=0,s_even=0;

public:
void get(){
for(int i=0;i<10;i++)
cin>>x[i];
}
```

```
OUTPUT

1
2
3
4
5
6
7
8
9
10

Odd Sum=5
Even Sum=5
```

```
void put(){
for(int i=0;i<10;i++){
if(x[i]%2==0)s_odd++;
else s_even++;
}
cout<<"Odd Sum="<<s_odd<<"\nEven
Sum="<<s_even;
}
}ob;

int main(){
ob.get();
ob.put();
}
```

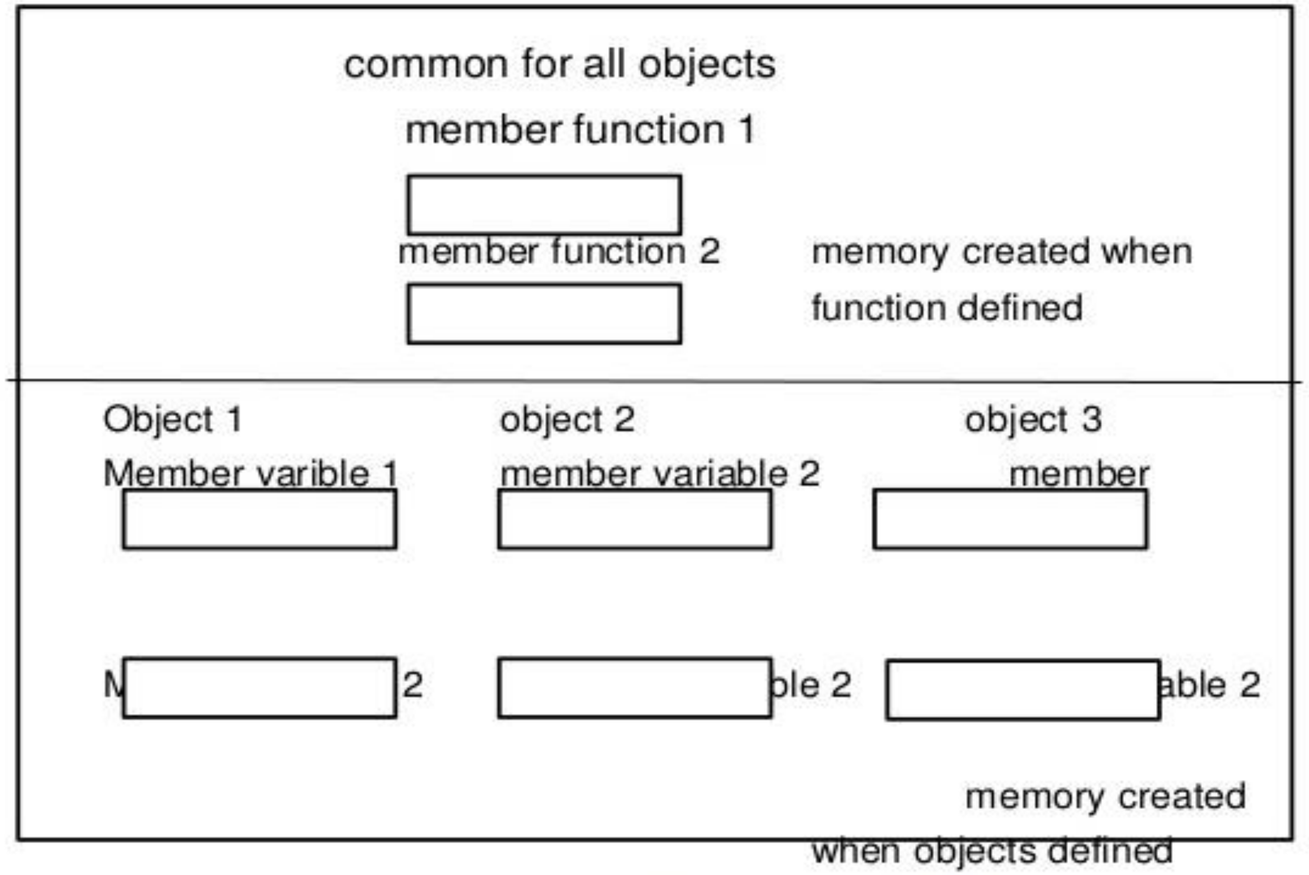
## Code

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

class odd_even{
int x[10];
int s_odd,s_even;

public:
void get(){
for(int i=0;i<10;i++)
cin>>x[i];
}
```

```
OUTPUT

1
2
3
4
5
6
7
8
9
10

Odd Sum=5
Even Sum=5
```

```
void put(){
for(int i=0;i<10;i++){
if(x[i]%2==0)s_odd++;
else s_even++;
}
cout<<"Odd Sum="<<s_odd<<"\nEven
Sum="<<s_even;
}
}ob;

int main(){
ob.get();
ob.put();
}
```

## Memory allocation to objects



## Code

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

class odd_even{
int x;
public:
void get(){
cout<<"Enter the value of x";
cin>>x;
}
int put(){
return x;
}
}ob[2];
```

```
int main()
{
ob[1].get();
ob[2].get();

int z=ob[1].put()+ob[2].put();
cout<<"Result="<<z;
}
```

OUTPUT

```
Enter the value of x10
Enter the value of x20
Result=30
```

## Code

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

class odd_even{
int x;

public:
void get(){
cout<<"Enter the value of x";
cin>>x;
}
int put(odd_even obj){
int z=x+obj.x;
return z;
}
}ob[2];

int main()
{
ob[1].get();
ob[2].get();

int z1=ob[1].put(ob[2]);
cout<<"Result="<<z1;
}
```

OUTPUT

```
Enter the value of x10
Enter the value of x20
Result=30
```

## Code

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

class odd_even{
int x;

public:
void get(){
cout<<"Enter the value of x";
cin>>x;
}

odd_even put(odd_even obj2){
odd_even z;
z.x=x+obj2.x;
return z;
}
void disp(){
cout<<x;
}
}ob[3];

int main()
{
ob[1].get();
ob[2].get();

ob[3]=ob[1].put(ob[2]);

ob[1].disp();
ob[2].disp();
ob[3].disp();
}
```

OUTPUT

```
Enter the value of x10
Enter the value of x20
102030
```

## Code

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

namespace alpha{
float y=2.5;
void put(){
cout<<"y="<<y;
}}

namespace beta{
float y=10.6;
void put(){
cout<<"y="<<y;
}}
```

```
int main(){
alpha::put();
beta::put();
}
```

OUTPUT

```
y=2.5
y=10.6
```