4장 매개변수와 오버로딩

4.0 참조 변수와 포인터 변수

* 참조 변수 (참조자) - C++의 새로운 특징

- 참조 변수는 그 자체로서 메모리를 차지하는 것이 아니라, 기존 변수에 또 다른 이름(alias, 별명)을 붙여서 해당 변수를 접근하는 방식이다.

- 선언 시 변수명 앞에 & 기호를 붙이고, 동시에 기존 특정 변수명으로 초기화되어야 한다.



|  |
| --- |
| int var1 = 10, var2 = 20;  int &ref1 = var1; // 정상  int &ref2; // 오류: 초기화 안됨  int &ref1 = var1; &ref1 = var2; // 오류: 대상 변경 불가  int &ref2 = var1+var2; // 오류: 수식  int &ref2 = 3; // 오류: 값 |

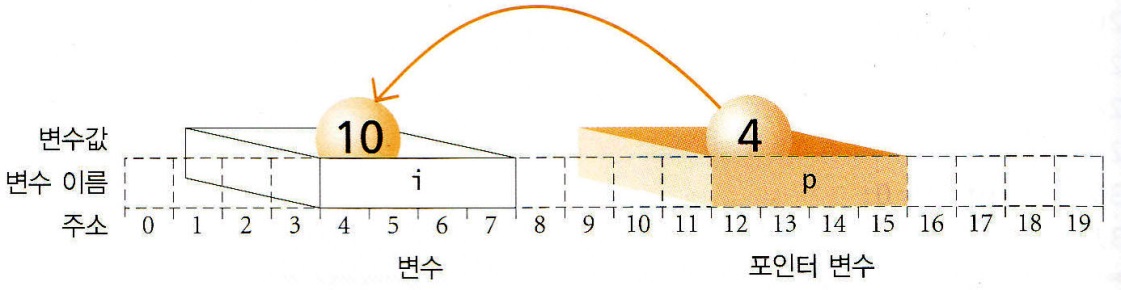
- **참조 변수는 자신만의 메모리 공간이 할당되지 않아, 가리키는 대상을 변경시킬 수 없다.** 그러나 가리키는 대상의 값은 변경 가능하다.

- 참조 변수는 값이나 수식에 대한 참조는 불가능하다.

- 참조 변수는 함수의 호출 사이에서 매우 유용하다.

* 포인터 변수

- 포인터 변수는 다른 변수의 주소를 저장한다.



|  |
| --- |
| int i = 10, j = 20;  int \*p; // 포인터 변수  p = &i;  cout << \*p; // 10 출력  p = &j;  cout << \*p; // 20 출력 |

- 주소 연산자(&)를 사용하여 특정 변수의 주소를 저장한다.

- 간접 참조 연산자(\*)를 이용하여 포인터가 가리키는 위치(주소)의 값을 읽어오거나 변경할 수 있다.

- 포인터는 변수로서 포인터가 가리키는 대상을 변경시킬 수 있다.

* 주소 연산자(&)의 차이 – 주의점

&ref1 = var1;

- var1의 주소에 또 다른 이름(별명) ‘ref1’을 붙이라는 의미이며, 선언문이나 함수의 헤더 에서 사용되지, 실행문에서는 사용되지 않는다.

p = &var1;

- var1의 주소를 p라는 변수에 저장하라는 의미이며, p는 주소를 저장할 수 있는 **포인터 변수**이어야 한다. 선언문이나 실행문에 다 사용된다.

* 사용법

참조 변수

- 참조 변수: 가리키는 대상이 변경되지는 않지만 해당 변수의 값을 바꾸고자 하는 경우 사용하며, 포인터 변수보다 쉽고 깔끔하다.

포인터 변수

- 가리키는 대상이 수시로 변경되는 경우 사용하며, NULL이 될 가능성이 있는 경우에도 사용한다.

예제: 참조 변수, 포인터 변수

ex04\_01.cpp

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {  //참조 변수  int var1 =10, var2 = 20;  int& ref1 = var1;  cout << var1 << endl;  cout << ref1 << endl;  // int &ref2; // 오류  // &ref1 = var2; // 오류  // int &ref2 = varl+var2; // 오류  // int &ref2 = 3; //오류  // 포인터 변수  int\* p;  p = &var1;  cout << \*p << endl;  p = &var2;  cout << \*p << endl;  return 0;  } |

4.1 매개변수

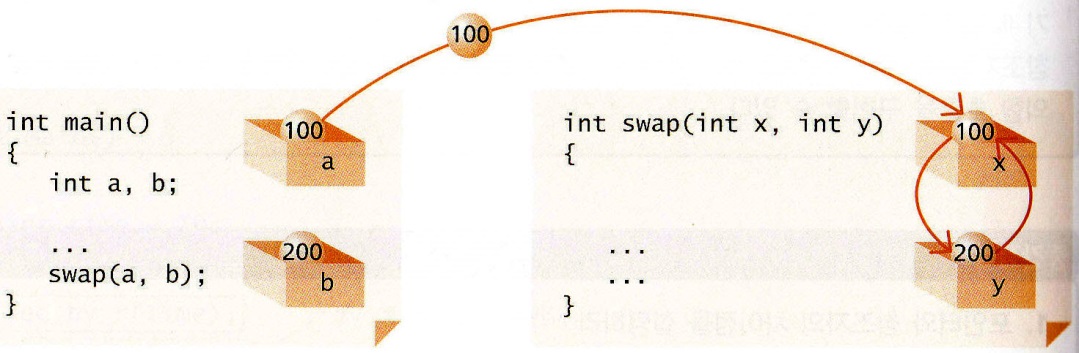
* 인자와 매개변수

- 함수 인자(argument): 함수를 호출할 때 사용되는 **변수 또는 인수** 내용물

sqrt(9.0); x=9; sqrt(x);

- 함수 매개변수(parameter, formal parameter, 형식 매개변수, 형식 인자): 함수가 수행되는 곳에 있으며 **인자를 대신하는(받아 들이는) 변수** 그릇

다음 예제에서 인자와 매개변수는?



|  |
| --- |
| int x = a;  int y = b; |

함수 호출

- 인자를 매개변수에 할당하는 방법

- 매개변수는 함수의 헤더에서 선언되어 있으므로, 함수의 본체에서 다시 한번 선언하지 않는다.

* call-by-value 호출(매개변수)

- 매개변수에 **인자의 값**만 할당(복사)된다.

- 매개변수는 지역변수로 인자 값으로 초기화 된다.

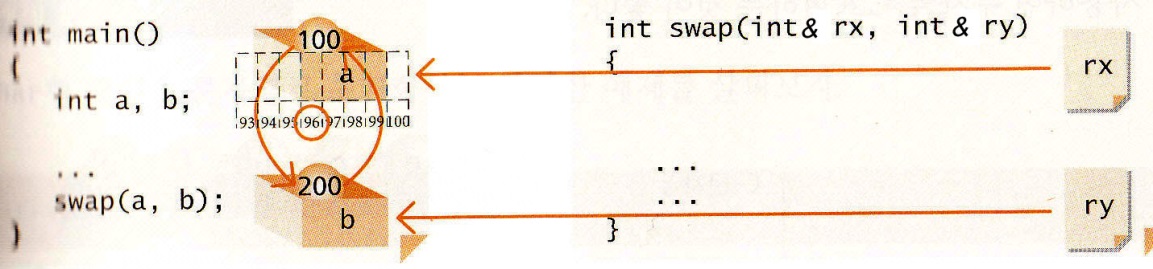
예제: call-by-value

ex04\_02.cpp

|  |
| --- |
| #include<iostream>  using namespace std;  void print\_data(int nun1, int num2);  int main()  {  int first, second;  cout << "Enter two numbers: ";  cin >> first >> second;  cout << "Before swap: " << first << " " << second << endl;    int temp;  temp = first;  first = second;  second = temp;  //cout << "After swap: " << first << " " << second << endl;  print\_data(first, second);  return 0;  }  void print\_data(int f, int s)  {  cout << "After swap: " << f << " " << s << endl;  } |

* call-by-reference 호출(매개변수) - C++의 특징

- 인자를 변수로 인식하여 **인자변수**라고 하자. 이 인자변수에 매개변수(별명)를 붙여서 인자 변수에 접근한다.



|  |
| --- |
| int &rx = a;  int &ry = b; |

- 함수 호출 시 인자는 변수여야만 하고 상수나 식은 안 된다.

- 역으로 생각하면, 함수 정의에서 매개변수가 인자변수로 대체되어 매개변수의 값을 바꾸면 인자변수의 값을 바꿀 수 있다.

- 함수 선언과 헤더에 있는 매개변수 형식에 &를 덧붙여서 표시한다. 함수 호출 시에는 &를 붙이지 않는다.





- 상수 call-by-reference 매개변수: 매개변수형 앞에 const를 붙이면 상수화되어 값을 변경할 수 없다. 배열이나 클래스형 매개변수에서 필요하다.

예제: call-by-reference

ex04\_03.cpp

|  |
| --- |
| #include<iostream>  using namespace std;  void print\_data(int nun1, int num2);  void swap(int& data1, int& data2); //void swap(int& data1, int& data2)  int main()  {  int first, second;  cout << "Enter two numbers: ";  cin >> first >> second;  cout << "Before swap: " << first << " " << second << endl;    swap(first, second);  print\_data(first, second);  return 0;  }  void print\_data(int f, int s)  {  cout << "After swap: " << f << " " << s << endl;  }  void swap(int &data1, int &data2) //int &data1 = first; int &data2 = second;  {  int temp;  temp = data1;  data1 = data2;  data2 = temp;  } |

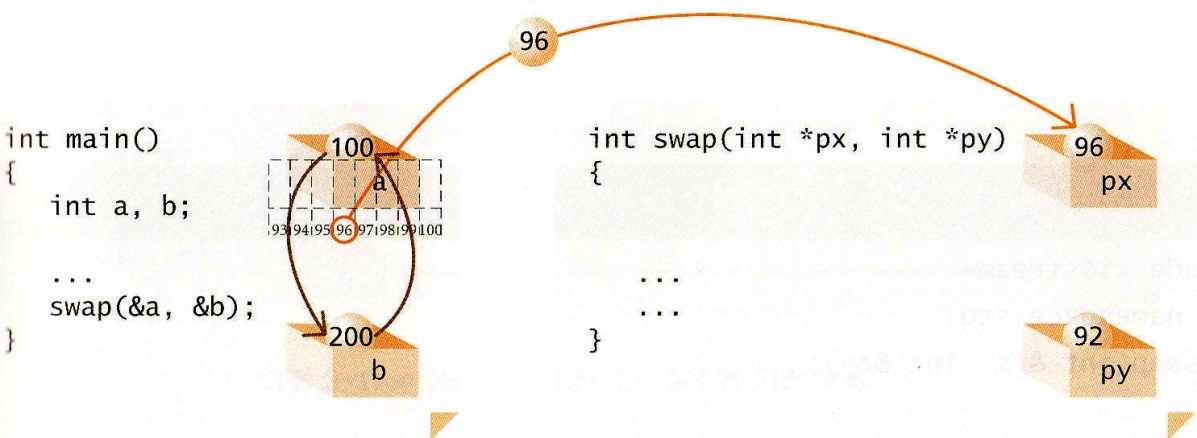
* call-by-pointer 호출(매개변수)

- 인자는 변수의 주소값을 매개변수에 보내고, 매개변수는 주소값을 받을 수 있는 포인터 변수로 선언한다.

- 함수의 본체에서는 매개 변수의 주소값을 변경할 수 있을 뿐만 아니라, 간접 참조 연산자(\*)을 통하여 기존 변수의 값을 접근할 수 있다.

- 함수 호출 시 반드시 주소값을 보내야 하며, 함수 본체에서 연산자(\*)를 생략하면 값이 아니라 주소값이 변경되어 원하지 않은 결과를 얻을 수 있다.

- 반면에, call-by-reference는 인자변수의 값이 변경되지, 주소값이 변경될 염려는 없다.



|  |
| --- |
| int \*px = &a;  int \*py = &b; |

예제: call-by-pointer

ex04\_04.cpp

|  |
| --- |
| #include<iostream>  using namespace std;  void get\_data(int\* v1, int\* v2); //void get\_data(int \*v1, int \*v2)  void swap(int &data1, int &data2);  void print\_data(int f, int s);  int main()  {  int first, second;  get\_data(&first, &second);  swap(first, second);  print\_data(first, second);  return 0;  }  void get\_data(int \*v1, int \*v2)  {  cout << "Enter two numbers: ";  cin >> \*v1 >> \*v2;  cout << "Before swap: " << \*v1 << " " << \*v2 << endl;  }  void swap(int &data1, int &data2) //int &data1 = first; int &data2 = second;  {  int temp;  temp = data1;  data1 = data2;  data2 = temp;  }  void print\_data(int f, int s)  {  cout << "After swap: " << f << " " << s << endl;  } |

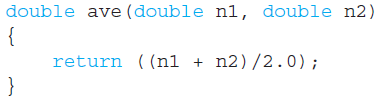
4.2 오버로딩과 디폴트 인자

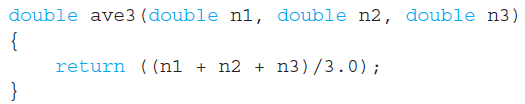
* 오버로딩(overloading)

- 같은 함수 이름으로 두 가지 이상의 함수를 정의할 수 있다.

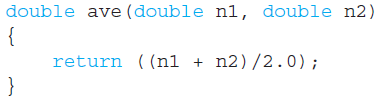
- 예제: 평균 구하기

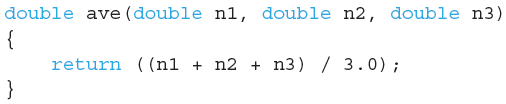
* + - 기존 C





* + - C++





- 각 함수는 자신의 선언(원형)을 가진다.

- 컴파일러는 함수 호출 시, **인자(매개 변수)의 수와 형**을 확인하여 구분한다.

- 오버로딩 할 수 없는 경우

* + - 반환되는 값의 형만 다른 함수들은 오버로딩 할 수 없다.
    - Const, 참조 연산(&), 포인터 연산(\*)에 의해서만 오버로딩 할 수 없다.

- 오버로딩은 자동 형 변환의 의미를 침해할 수 있다. 즉, 오버로딩이 자동 형 변환보다 우선 시 된다.

오버로딩 규칙

1. 정확한 일치: 인자의 개수와 형이 정확하게 정의와 맞는다면 그 정의를 선택한다.

2. 자동 형 변환을 사용한 일치: 만일 정확하게 일치하지 않지만 자동 형 변환을 사용하여 일치된다면 그 정의를 사용한다.

  m ← 99.0

- 오류

예제: 오버로딩

ex04\_05.cpp

|  |
| --- |
| #include<iostream>  using namespace std;  void get\_data(int& v1, int& v2, double& v3);  // function overloading  double avg(int num1, int num2);  double avg(int num1, double num2);  double avg(int num1, int num2, double num3);  int main()  {  int first, second;  double third;  get\_data(first, second, third);  // function overloading  cout << "The mean of two numbers = " << avg(first, second) << endl;  cout << "The mean of two numbers = " << avg(first, third) << endl;  cout << "The mean of three numbers = " << avg(first, second, third) << endl;    return 0;  }  void get\_data(int &v1, int &v2, double &v3)  {  cout << "Enter three numbers: ";  cin >> v1 >> v2 >> v3;  }  double avg(int num1, int num2)  {  return ((num1 + num2) / 2.0);  }  double avg(int num1, double num2)  {  return ((num1 + num2) / 2.0);  }  double avg(int num1, int num2, double num3)  {  return ((num1 + num2 + num3) / 3.0);  } |

* 디폴트 인자

- 한 함수에서 하나 이상의 **call-by-value 매개변수**에 대해 디폴트 인자를 정의할 수 있다.

- 만일 대응하는 인자가 생략되면 디폴트 인자 값으로 대체된다.

- 디폴트 인자는 함수 선언에서 주어지며 함수 정의에서 주어지지 않는다.

- 모든 디폴트 인자의 위치는 가장 오른쪽에 있어야 한다.

예제: 디폴트 연산자

ex04\_06.cpp

|  |
| --- |
| #include<iostream>  using namespace std;  void get\_data(int& v1, int& v2, double& v3);  // function overloading  double sum(int num1, int num2=0, double num3=0.0);  int main()  {  int first, second;  double third;  get\_data(first, second, third);  // function overloading  cout << "The sum of one number = " << sum(first) << endl;  cout << "The sum of two numbers = " << sum(first, second) << endl;  cout << "The sum of three numbers = " << sum(first, second, third) << endl;    return 0;  }  void get\_data(int &v1, int &v2, double &v3)  {  cout << "Enter three numbers: ";  cin >> v1 >> v2 >> v3;  }  double sum(int num1, int num2, double num3)  {  return (num1 + num2 + num3) ;  } |

4.3 테스팅과 디버깅 함수

* assert 매크로

- 프로그램상 어떤 가정을 점검하기 위해서 사용한다. 즉, 항상 true가 되는 조건식을 만들고 조건식이 false가되면, assert 매크로는 오류가 있음을 알린다.

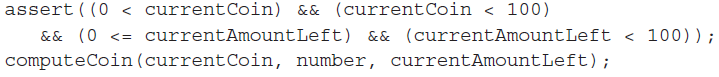
- assert 매크로는 bool형의 call-by-value 매개변수를 가지는 void함수처럼 사용된다.

- #include <cassert>에 있다.

- 예제

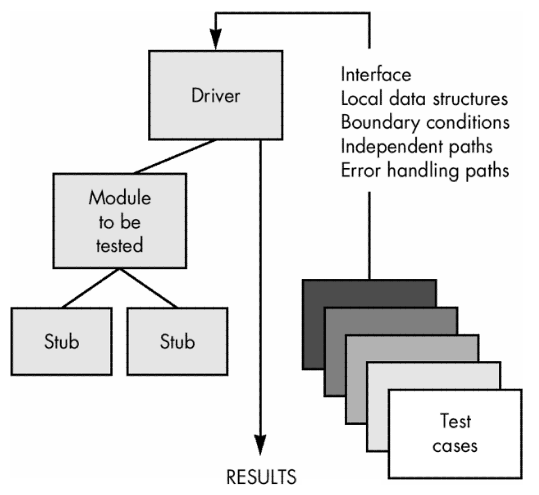


* + - assert 매크로



- assert 매크로 실행 중단은 #define NDEBUG 을 #include<cassert> 앞에 추가한다. 재실행은 주석처리 또는 삭제를 하면 된다.

* 스터브와 드라이버



드라이버 프로그램(driver program)

- 어떤 함수를 테스트하기 위해 그 함수가 사용될 임시 프로그램을 의미한다.

- 드라이버 프로그램은 작고 단순할수록 좋다.

스터브(stub)

- 아직 작성되지 않은 함수나 테스트되지 않는 함수를 테스트하는 것은 불가능하다. 하지만 빠진 함수나 테스트되지 않은 함수의 단순화된 버전을 사용할 수 있다.

- 이런 단순화된 함수를 ‘스터브’라고 한다.

- 스터브는 전체적인 프로그램의 개요를 작성하는데 효율적이다.